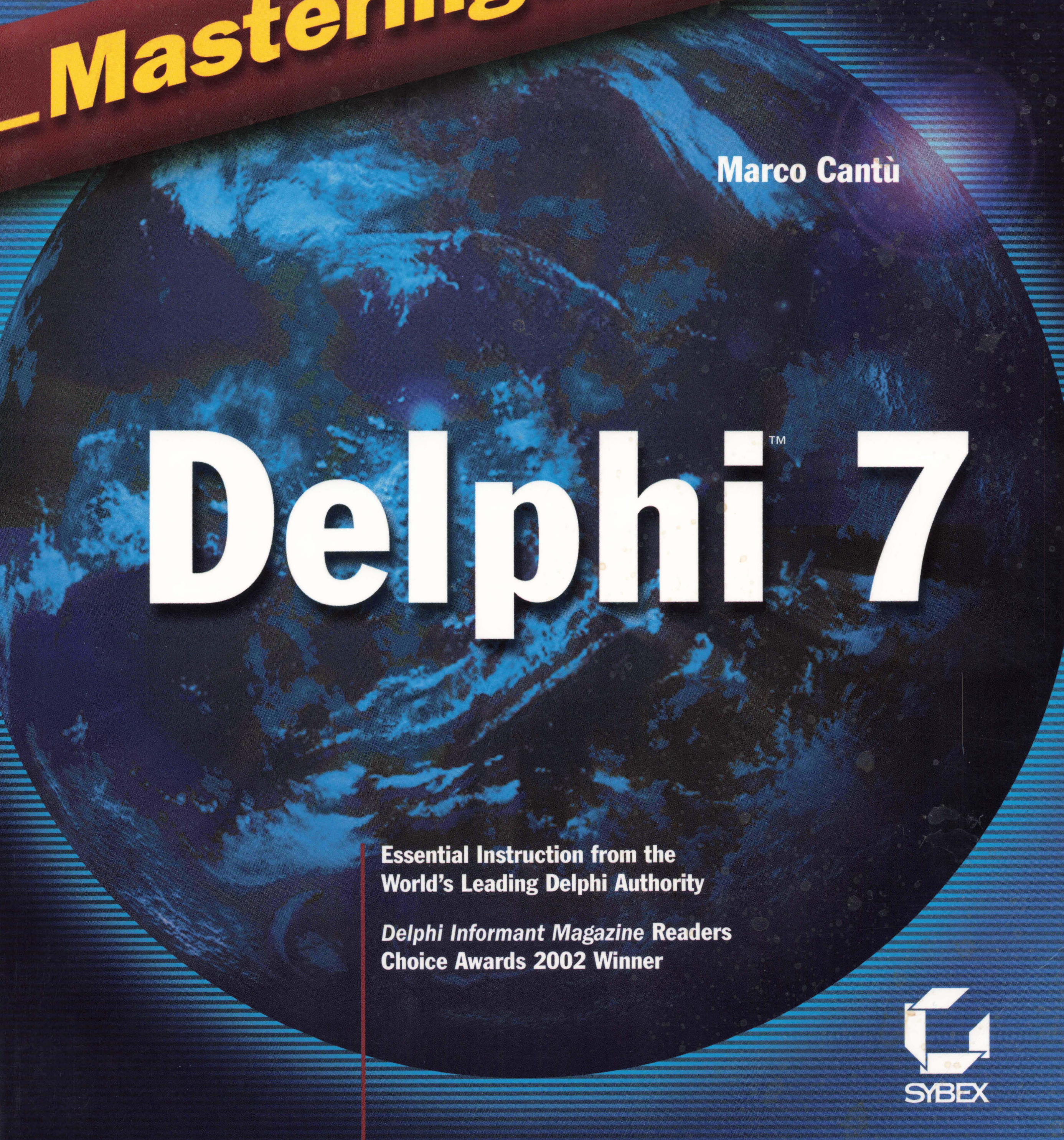


Mastering™

Marco Cantù

Delphi™ 7

Essential Instruction from the World's Leading Delphi Authority

***Delphi Informant Magazine* Readers Choice Awards 2002 Winner**

SYBEX

New Delphi 7 Features

FEATURE	**CHAPTER**
Extended Compiler Messages and Search Results	1
Extended String Formatting Routines	3
The PosEx function	3
TStringList extensions	4
Compression Stream Classes	4
Windows XP Themes	6
Snapping a Form to the Screen	7
ModelMaker	11
UML Diagrams	11
COM/.NET interoperability	12
DbExpress driver for Microsoft SQL Server	14
SimpleDataSet component	14
Rave	18
Printing Reports	18
Indy 9	19
Updated Web App Debugger	20
IntraWeb Web Applications	21
IntraWeb Client-Side Components	21
OpenXML	22
Updated XSLT support in WebSnap	22
Updated WSDL publishing	23
IAppServerSOAP interface	23
SOAP Attachments	23
UDDI	23
Delphi for .NET Support	24
Delphi for .NET Compiler Preview	24 and 25
Delphi for .NET Language Changes	25
.NET RTL Compiler and Libraries	25
VCL.NET	25

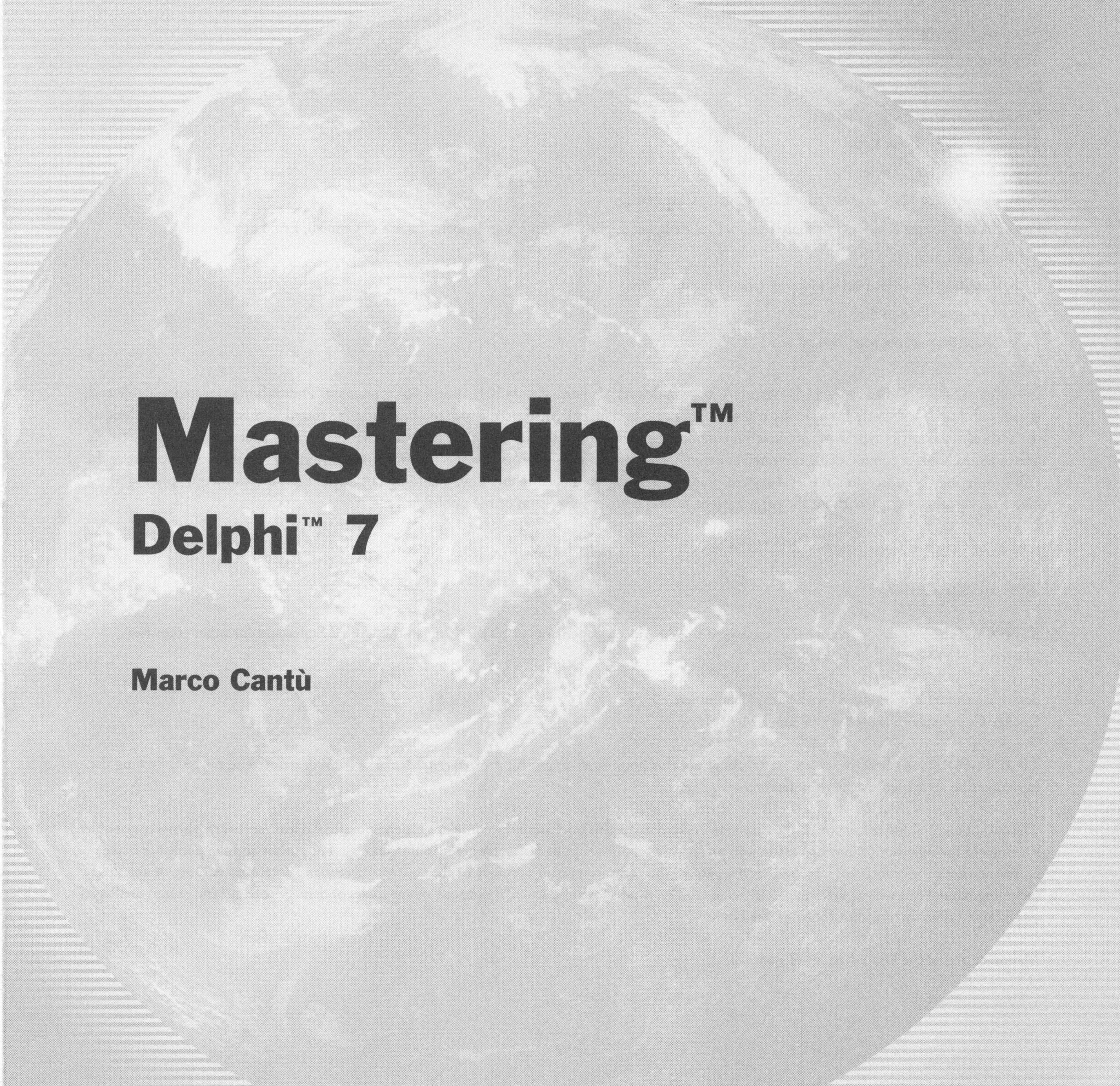
Praise for the Previous Edition of *Mastering Delphi*, Winner of the *Delphi Informant Magazine*'s 2002 Readers Choice Award for Best Book

"...its scope is immense, its treatment of Delphi exhaustive, and it's extremely valuable as a reference. If you're new to Delphi programming and are ready for more than what the in-the-box manuals can provide, or if you want a general-purpose Delphi book, *Mastering Delphi 6* more than fills the bill."
—*Delphi Informant Magazine*

"Marco has produced the usually high-standard best-selling quality in *Mastering Delphi 6* (you won't be sorry if you read this book, even if you read the previous edition as well)..."
—*Bob Swart, Dr. Bob's Programming Clinic (www.drbob42.com)*

"Whether you class yourself as just a beginner or have long been a lover of Borland's much-acclaimed object-oriented development tool, Mastering Delphi 6 has plenty to offer you. Written by world-renowned Delphi expert Marco Cantù, this edition, marking the release of Delphi 6, continues the tradition of excellence, balancing attention to detail with the author's unfussy, very readable style, which makes this a book that's accessible to all.... Designed in a tutorial format, this book bristles with practical programming. In all, there are around 300 examples, each one supported by Cantù's clear explanation of the key skills in view. From the foundations of Delphi's object-oriented class library through to a whole section devoted to building Web applications, no information is left unexplored. If you are looking to lever your skills and harness the power of Delphi 6 for your applications, look no further than this excellent resource."
—*Peter Lunn, Amazon.com Official Review*
Courtesy of Amazon.com

"*Mastering Delphi 6* is the completely revised and updated edition of the industry's favorite Delphi programming book, and offers the most complete coverage of Delphi 6 programming available anywhere.... The book is presented clearly and attractive; practical, tutorial-based coverage helps you develop key skills, solve tough problems, and build and implement sophisticated Windows applications.... Since this is a book review, I'm supposed to find some book weaknesses. I simply can't."
—*Zarko Gajic, About.com's Delphi expert*



Mastering™

Delphi™ 7

Marco Cantù

SYBEX®
San Francisco London

Associate Publisher: Joel Fugazzotto
Acquisitions Editor: Denise Santoro Lincoln
Developmental Editor: Brianne Agatep
Production Editor: Kelly Winquist
Technical Editor: Brian Long
Copyeditor: Tiffany Taylor
Compositor: Rozi Harris, Interactive Composition Corporation
Proofreaders: Nancy Riddiough, Emily Hsuan, Leslie Higbee Light, Monique Vandenberg, Laurie O'Connell, Eric Lach
Indexer: Ted Laux
Book Designer: Maureen Forys, Happenstance Type-O-Rama
Cover Designer: Design Site
Cover Illustrator: Tania Kac, Design Site

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. The author(s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2002115474

ISBN: 0-7821-4201-X

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.
Mastering is a trademark of SYBEX Inc.

Screen reproductions produced with Collage Complete.
Collage Complete is a trademark of Inner Media Inc.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2

To the late Andrea Gnesutta,
a friend the Italian Delphi community prematurely lost.

Acknowledgments

This seventh edition of *Mastering Delphi* follows the seventh release of a Delphi development environment by Borland, a revolution started in the winter of year 1994. As it has for many other programmers, Delphi (and its Linux twin, Kylix) has been my primary interest throughout these years; and writing, consulting, teaching, and speaking at conferences about Delphi have absorbed more and more of my time, leaving other languages and programming tools in the dust of my office. Because my work and my life are quite intertwined, many people have been involved in both, and I wish I had enough space and time to thank them all as they deserve. Instead, I'll just mention a few particular people and say a warm "Thank You" to the entire Delphi community (especially for the Spirit of Delphi 1999 Award I've been happy to share with Bob Swart).

The first official thanks are for the Borland programmers and managers who made Delphi possible and continue to improve it: Chuck Jazdzewski, Danny Thorpe, Eddie Churchill, Allen Bauer, Steve Todd, Mark Edington, Jim Tierney, Ravi Kumar, Jörg Weingarten, Anders Ohlsson, and all the others I have not had a chance to meet. I'd also like to give particular thanks to my friends John Kaster and David Intersimone (at Borland's Developer Relations), and others who have worked at Borland, including Charlie Calvert and Zack Urlocker.

The next thanks are for the Sybex editorial and production crew, many of whom I don't even know. Special thanks go to Brianne Agatep, Denise Santoro Lincoln, Tiffany Taylor, Rozi Harris, and Kelly Winquist; I'd also like to thank Joel Fugazzotto and Monica Baum.

This edition of *Mastering Delphi* has had a very detailed and scrupulous review from Delphi guru Brian Long (`www.blong.com`). His highlights and comments have improved the book in all areas: technical content, accuracy, examples, and even readability and grammar! Thanks a lot. In writing this book I had special contributions (to different extents) to the chapters on add-on tools and in the area of .NET programming from (in alphabetical order) John Bushakra, Jim Gunkel, Chad Hower, and Robert Leahey. A short bio and contact information for each of them is in the chapters they helped me write.

Previous editions also had special contributions: Tim Gooch worked on *Mastering Delphi 4* and Giuseppe Madaffari contributed database material for the Delphi 5 edition. For *Mastering Delphi 6*, Guy Smith-Ferrier rewrote the chapter on ADO and Nando Dessena helped me with the InterBase material. Many improvements to the text and sample programs were suggested by technical reviewers of past editions (Delphi R&D team member Danny Thorpe, Juancarlo Añez, Ralph Friedman, Tim Gooch, and Alain Tadros) and in other reviews over the years by Bob Swart, Giuseppe Madaffari, and Steve Tendon. Uberto Barbini helped me write *Mastering Kylix 2* and some of his ideas ended up also affecting this book.

Special thanks go to my friends Bruce Eckel, Andrea Provaglio, Norm McIntosh, Johanna and Phil of the BUG-UK, Ray Konopka, Mark Miller, Cary Jensen, Chris Frizelle of *The Delphi Magazine*, Mike Orriss, Dan Miser, my co-worker Paolo Rossi, and the entire Italian D&D Team (`www.dedonline.com`). Also, a very big "Thank You" to all the attendees of my Delphi programming courses, seminars, and conferences in Italy, the United States, France, the United Kingdom, Singapore, the Netherlands, Germany, Sweden….

My biggest thanks go to my wife Lella who had to endure yet another book-writing session and too many late nights (after spending the evenings with our daughter, Benedetta—I'll thank her with a hug, as Daddy's book looks quite boring to her). Many of our friends (and their kids) provided healthy breaks in the work: Sandro and Monica with Luca, Stefano and Elena, Marco and Laura with Matteo and Filippo, Bianca and Paolo, Luca and Elena with Tommaso, Chiara and Daniele with Leonardo and Matteo, Laura, Vito and Marika with Sofia. Our parents, brothers, sisters, and their families were very supportive, too. It was nice to spend some of our free time with them and our seven nephews—Matteo, Andrea, Giacomo, Stefano, Andrea, Pietro, and Elena.

Finally, I would like to thank all of the people, many of them unknown, who enjoy life and help to build a better world. If I never stop believing in the future and in peace, it is also because of them.

Contents

Introduction . . . . *xxv*

Part I • Foundations **1**

Chapter 1 • Delphi 7 and Its IDE **3**

Editions of Delphi 3
An Overview of the IDE 4
    An IDE for Two Libraries 5
    Desktop Settings 5
    Environment Options 6
    About Menus 7
    The Environment Options Dialog Box 7
    The To-Do List 7
    Extended Compiler Messages and Search Results in Delphi 7 9
The Delphi Editor 10
    The Code Explorer 11
    Browsing in the Editor 12
    Class Completion 13
    Code Insight 14
    More Editor Shortcut Keys 16
    Loadable Views 16
The Form Designer 18
    The Object Inspector 19
    The Object TreeView 22
Secrets of the Component Palette 23
    Copying and Pasting Components 24
    From Component Templates to Frames 25
Managing Projects 26
    Project Options 28
    Compiling and Building Projects 30
    Exploring a Project's Classes 32
Additional and External Delphi Tools 33
The Files Produced by the System 34
    Looking at Source Code Files 38
The Object Repository 39
Debugger Updates in Delphi 7 42
What's Next? 42

Chapter 2 • The Delphi Programming Language **43**

Core Language Features 44
Classes and Objects 44
    More on Methods 46
    Creating Components Dynamically 47

Encapsulation 48
  Private, Protected, and Public 48
  Encapsulating with Properties 50
  Encapsulation and Forms 52
Constructors 54
  Destructors and the *Free* Method 55
Delphi's Object Reference Model 55
  Assigning Objects 56
  Objects and Memory 58
Inheriting from Existing Types 59
  Protected Fields and Encapsulation 60
  Inheritance and Type Compatibility 62
Late Binding and Polymorphism 63
  Overriding and Redefining Methods 64
  Virtual versus Dynamic Methods 65
  Abstract Methods 66
Type-Safe Down-Casting 67
Using Interfaces 69
Working with Exceptions 71
  Program Flow and the *finally* Block 72
  Exception Classes 73
Logging Errors 75
Class References 76
  Creating Components Using Class References 77
What's Next? 79

Chapter 3 • The Run-Time Library 81

The Units of the RTL 81
  The System and SysInit Units 84
  The SysUtils and SysConst Units 85
  The Math Unit 88
  The ConvUtils and StdConvs Units 91
  The DateUtils Unit 91
  The StrUtils Unit 91
  The Types Unit 93
  The Variants and VarUtils Units 94
  The DelphiMM and ShareMem Units 95
  COM-Related Units 96
Converting Data 96
What About Currency Conversions? 99
Managing Files with SysUtils 102
The *TObject* Class 104
  Showing Class Information 107
What's Next? 108

Chapter 4 • Core Library Classes 109

The RTL Package, VCL, and CLX 110
  Traditional Sections of VCL 110

The Structure of CLX 111
VCL-Specific Sections of the Library 112
The *TPersistent* Class 112
The *published* Keyword 114
Accessing Properties by Name 115
The *TComponent* Class 117
Ownership 117
The *Name* Property 121
Removing Form Fields 121
Hiding Form Fields 122
The Customizable Tag Property 124
Events 124
Events in Delphi 124
Method Pointers 125
Events Are Properties 126
Lists and Container Classes 128
Lists and String Lists 128
Collections 131
Container Classes 131
Type-Safe Containers and Lists 133
Streaming 135
The *TStream* Class 135
Specific Stream Classes 137
Using File Streams 138
The *TReader* and *TWriter* Classes 139
Streams and Persistency 140
Compressing Streams with ZLib 145
Summarizing the Core VCL and BaseCLX Units 146
The Classes Unit 147
Other Core Units 148
What's Next? 148

Chapter 5 • Visual Controls 149

VCL versus VisualCLX 149
Delphi's Dual Library Support 151
Choosing a Visual Library 155
Converting Existing Applications 156
TControl and Derived Classes 158
Parent and Controls 158
Properties Related to Control Size and Position 159
Activation and Visibility Properties 159
Fonts 159
Colors 160
The *TWinControl* Class (VCL) 161
The *TWidgetControl* Class (CLX) 161
Opening the Component Toolbox 162
The Text Input Components 162
Selecting Options 165

- Lists . . . 167
- Ranges . . . 172
- Commands . . . 173

Control-Related Techniques . . . 176
- Handling the Input Focus . . . 176
- Control Anchors . . . 179
- Using the Splitter Component . . . 180
- Accelerator Keys . . . 181
- Using the Fly-by Hints . . . 182
- Owner-Draw Controls and Styles . . . 184

ListView and TreeView Controls . . . 188
- A Graphical Reference List . . . 188
- A Tree of Data . . . 193
- Custom Tree Nodes . . . 197

What's Next? . . . 199

Chapter 6 • Building the User Interface . . . 201

Multiple-Page Forms . . . 202
- PageControls and TabSheets . . . 202
- An Image Viewer with Owner-Draw Tabs . . . 207
- The User Interface of a Wizard . . . 210

The ToolBar Control . . . 213
- The RichBar Example . . . 214
- A Menu and a Combo Box in a Toolbar . . . 215
- A Simple Status Bar . . . 217

Themes and Styles . . . 219
- CLX Styles . . . 219
- Windows XP Themes . . . 220

The ActionList Component . . . 222
- Predefined Actions in Delphi . . . 224
- Actions in Practice . . . 225
- The Toolbar and ActionList of an Editor . . . 229

Toolbar Containers . . . 230
- The ControlBar . . . 231
- Delphi's Docking Support . . . 234
- Docking Toolbars in ControlBars . . . 235
- Docking to a PageControl . . . 239

The ActionManager Architecture . . . 241
- Building a Simple Demo . . . 242
- Least-Recently Used Menu Items . . . 245
- Porting an Existing Program . . . 247
- Using List Actions . . . 248

What's Next? . . . 250

Chapter 7 • Working with Forms . . . 251

The *TForm* Class . . . 251
- Using Plain Forms . . . 252
- The Form Style . . . 254

    - The Border Style 254
    - The Border Icons 256
    - Setting More Window Styles 257
- Direct Form Input 259
    - Supervising Keyboard Input 259
    - Getting Mouse Input 261
    - Dragging and Drawing with the Mouse 262
- Painting on Forms 265
- Unusual Techniques: Alpha Blending, Color Key, and the Animate API 267
- Position, Size, Scrolling, and Scaling 268
    - The Form Position 268
    - Snapping to the Screen (in Delphi 7) 269
    - The Size of a Form and Its Client Area 269
    - Form Constraints 270
    - Scrolling a Form 270
    - Scaling Forms 274
    - Automatic Form Scaling 276
- Creating and Closing Forms 277
    - Form Creation Events 278
    - Closing a Form 279
- Dialog Boxes and Other Secondary Forms 280
    - Adding a Second Form to a Program 280
    - Creating Secondary Forms at Run Time 281
- Creating a Dialog Box 282
    - The Dialog Box of the RefList Example 283
    - A Modeless Dialog Box 285
- Predefined Dialog Boxes 287
    - Windows Common Dialogs 287
    - A Parade of Message Boxes 289
- About Boxes and Splash Screens 289
    - Building a Splash Screen 289
- What's Next? 292

Part II • Delphi Object-Oriented Architectures 293

Chapter 8 • The Architecture of Delphi Applications 295

- The *Application* Object 295
    - Displaying the Application Window 297
    - Activating Applications and Forms 298
    - Tracking Forms with the *Screen* Object 299
- From Events to Threads 302
    - Event-Driven Programming 302
    - Windows Message Delivery 303
    - Background Processing and Multitasking 304
    - Delphi Multithreading 304
- Checking for a Previous Instance of an Application 307
    - Looking for a Copy of the Main Window 307

  Using a Mutex 308
  Searching the Window List 308
  Handling User-Defined Window Messages 309
Creating MDI Applications 310
  MDI in Windows: A Technical Overview 310
Frame and Child Windows in Delphi 311
  Building a Complete Window Menu 312
  The MdiDemo Example 313
MDI Applications with Different Child Windows 315
  Child Forms and Merging Menus 315
  The Main Form 316
  Subclassing the MDI Client Window 317
Visual Form Inheritance 318
  Inheriting from a Base Form 319
  Polymorphic Forms 321
Understanding Frames 324
  Frames and Pages 326
  Multiple Frames with No Pages 328
Base Forms and Interfaces 330
  Using a Base Form Class 330
  Using Interfaces 333
Delphi's Memory Manager 334
What's Next? 336

Chapter 9 • Writing Delphi Components 337

Extending the Delphi Library 337
  Component Packages 338
  Rules for Writing Components 339
  The Base Component Classes 340
Building Your First Component 340
  The Fonts Combo Box 341
  Creating a Package 344
  Using the Font Combo Box 347
  The Component Palette Bitmaps 348
Creating Compound Components 349
  Internal Components 349
  Publishing Subcomponents 350
  External Components 352
  Referring to Components with Interfaces 354
A Complex Graphical Component 358
  Defining an Enumerated Property 359
  Writing the *Paint* Method 361
  Adding *TPersistent* Properties 362
  Defining a Custom Event 364
  Registering Property Categories 366
Customizing Windows Controls 368
  The Numeric Edit Box 368
  The Sound Button 371

  Handling Internal Messages: The Active Button ... 372
  Component Messages and Notifications ... 373
A Dialog Box in a Component ... 378
  Using the Nonvisual Component ... 381
Collection Properties ... 381
Defining Custom Actions ... 385
Writing Property Editors ... 388
  An Editor for the Sound Properties ... 388
  Installing the Property Editor ... 391
Writing a Component Editor ... 392
  Deriving from the *TComponentEditor* Class ... 393
  A Component Editor for the ListDialog ... 393
  Registering the Component Editor ... 395
What's Next? ... 395

Chapter 10 • Libraries and Packages ... 397

The Role of DLLs in Windows ... 397
  What Is Dynamic Linking? ... 397
  What Are DLLs For? ... 398
  Rules for Delphi DLL Writers ... 399
Using Existing DLLs ... 400
  Using a C++ DLL ... 401
Creating a DLL in Delphi ... 402
  Your First Delphi DLL ... 403
  Calling the Delphi DLL ... 406
Advanced Features of Delphi DLLs ... 407
  Changing Project and Library Names ... 407
  Calling a DLL Function at Run Time ... 408
  Placing Delphi Forms in a Library ... 410
Libraries in Memory: Code and Data ... 411
  Sharing Data with Memory-Mapped Files ... 413
Using Delphi Packages ... 415
  Package Versioning ... 416
Forms Inside Packages ... 417
  Loading Packages at Run Time ... 419
  Using Interfaces in Packages ... 420
The Structure of a Package ... 424
What's Next? ... 427

Chapter 11 • Modeling and OOP Programming (with ModelMaker) ... 429

Understanding ModelMaker's Internal Model ... 430
Modeling and UML ... 431
  Class Diagrams ... 431
  Sequence Diagrams ... 433
  Use Cases and Other Diagrams ... 434
  Non-UML Diagrams ... 435
  Common Diagram Elements ... 435

Coding Features of ModelMaker 436
  Delphi/ModelMaker Integration 437
  Managing the Code Model 438
  The Unit Code Editor 439
  The Method Implementation Code Editor 441
  The Difference View 442
  The Event Types View 443
Documentation and Macros 444
  Documentation versus Comments 444
  Working with Macros 445
Refactoring Your Code 446
  Applying Design Patterns 447
  Code Templates 450
Little-Known Tidbits 452
What's Next? 452

Chapter 12 • From COM to COM+ 455

A Short History of OLE and COM 456
Implementing *IUnknown* 457
  Globally Unique Identifiers 458
  The Role of Class Factories 460
A First COM Server 460
  COM Interfaces and Objects 461
  Initializing the COM Object 464
  Testing the COM Server 464
  Using Interface Properties 465
  Calling Virtual Methods 466
Automation 467
  Dispatching an Automation Call 468
Writing an Automation Server 470
  The Type-Library Editor 470
  The Server Code 472
  Registering the Automation Server 473
  Writing a Client for the Server 474
  The Scope of Automation Objects 476
  The Server in a Component 477
  COM Data Types 478
  Using Office Programs 478
Using Compound Documents 479
  The Container Component 480
  Using the Internal Object 482
Introducing ActiveX Controls 483
  ActiveX Controls Versus Delphi Components 484
  Using ActiveX Controls in Delphi 485
Writing ActiveX Controls 486
  Building an ActiveX Arrow 487
  Adding New Properties 488
  Adding a Property Page 489

ActiveForms 491
ActiveX in Web Pages 492
Introducing COM+ 494
Creating a COM+ Component 494
Transactional Data Modules 497
COM+ Events 497
COM and .NET in Delphi 7 500
What's Next? 502

Part III • Delphi Database-Oriented Architectures 503

Chapter 13 • Delphi's Database Architecture 505

Accessing a Database: dbExpress, Local Data, and Other Alternatives 506
The dbExpress Library 506
The Borland Database Engine 507
InterBase Express 507
MyBase and the ClientDataSet Component 508
dbGo for ADO 508
Custom Dataset Components 509
MyBase: Stand-alone ClientDataSet 509
Connecting to an Existing Local Table 509
From the Midas DLL to the MidasLib Unit 510
XML and CDS Formats 511
Defining a New Local Table 511
Indexing 512
Filtering 513
Locating Records 514
Using Data-Aware Controls 515
Data in a Grid 516
DBNavigator and Dataset Actions 516
Text-Based Data-Aware Controls 516
List-Based Data-Aware Controls 517
Using Lookup Controls 518
Graphical Data-Aware Controls 520
The DataSet Component 520
The Status of a Dataset 524
The Fields of a Dataset 525
Using Field Objects 527
A Hierarchy of Field Classes 529
Adding a Calculated Field 530
Lookup Fields 533
Handling Null Values with Field Events 535
Navigating a Dataset 536
The Total of a Table Column 537
Using Bookmarks 538
Editing a Table Column 539
Customizing a Database Grid 540
Painting a DBGrid 540

  - A Grid Allowing Multiple Selection 542
  - Dragging to a Grid 544
- Database Applications with Standard Controls 544
  - Mimicking Delphi Data-Aware Controls 545
  - Sending Requests to the Database 547
- Grouping and Aggregates 549
  - Grouping 549
  - Defining Aggregates 550
- Master/Detail Structures 552
  - Master/Detail with ClientDataSets 552
- Handling Database Errors 553
- What's Next? 554

Chapter 14 • Client/Server with dbExpress 555

- The Client/Server Architecture 555
- Elements of Database Design 557
  - Entities and Relations 557
  - From Primary Keys to OIDs 558
  - More Constraints 560
  - Unidirectional Cursors 560
- Introducing InterBase 561
  - Using IBConsole 562
  - InterBase Server-Side Programming 564
- The dbExpress Library 566
  - Working with Unidirectional Cursors 566
  - Platforms and Databases 567
  - Driver Versioning Troubles and Embedded Units 568
- The dbExpress Components 569
  - The SQLConnection Component 569
  - The dbExpress Dataset Components 572
  - The SQLMonitor Component 574
- A Few dbExpress Demos 574
  - Using a Single Component or Many Components 575
  - Accessing Database Metadata with *SetSchemaInfo* 578
  - A Parametric Query 579
  - When One-Way Is Enough: Printing Data 581
- The Packets and the Cache 584
  - Manipulating Updates 584
  - Updating the Data 586
  - Using Transactions 589
- Using InterBase Express 591
  - IBX Dataset Components 592
  - IBX Administrative Components 593
  - Building an IBX Example 593
  - Building a Live Query 594
  - Monitoring InterBase Express 598
  - Getting More System Data 599
- Real-World Blocks 600
  - Generators and IDs 600

- Case-Insensitive Searches . . . 602
- Handling Locations and People . . . 603
- Building a User Interface . . . 605
- Booking Classes . . . 608
- Building a Lookup Dialog . . . 611
- Adding a Free Query Form . . . 613

What's Next? . . . 614

Chapter 15 • Working with ADO . . . 615

Microsoft Data Access Components (MDAC) . . . 616
- OLE DB Providers . . . 617

Using dbGo Components . . . 618
- A Practical Example . . . 619
- The ADOConnection Component . . . 621
- Data Link Files . . . 621
- Dynamic Properties . . . 622
- Getting Schema Information . . . 622

Using the Jet Engine . . . 624
- Paradox through Jet . . . 624
- Excel through Jet . . . 625
- Text Files through Jet . . . 626
- Importing and Exporting . . . 628

Working with Cursors . . . 629
- Cursor Location . . . 629
- Cursor Type . . . 630
- Ask and Ye Shall Not Receive . . . 631
- No Record Count . . . 631
- Client Indexes . . . 632
- Cloning . . . 633

Transaction Processing . . . 634
- Nested Transactions . . . 634
- ADOConnection Attributes . . . 635
- Lock Types . . . 635

Updating the Data . . . 636
- Batch Updates . . . 638
- Optimistic Locking . . . 639
- Resolving Update Conflicts . . . 641

Disconnected Recordsets . . . 642
- Connection Pooling . . . 643
- Persistent Recordsets . . . 644
- The Briefcase Model . . . 645

A Word on ADO.NET . . . 645

What's Next? . . . 646

Chapter 16 • Multitier DataSnap Applications . . . 647

One, Two, Three Levels in Delphi's History . . . 647
- The Technical Foundation of DataSnap . . . 649
- The *IAppServer* Interface . . . 649

- The Connection Protocol 650
- Providing Data Packets 651
- Delphi Support Components (Client-Side) 652
- Delphi Support Components (Server-Side) 653

Building a Sample Application 653
- The First Application Server 653
- The First Thin Client 655

Adding Constraints to the Server 656
- Field and Dataset Constraints 657
- Including Field Properties 658
- Field and Table Events 658

Adding Features to the Client 659
- The Update Sequence 660
- Refreshing Data 660

Advanced DataSnap Features 662
- Parametric Queries 663
- Custom Method Calls 663
- Master/Detail Relations 664
- Using the Connection Broker 666
- More Provider Options 666
- The Simple Object Broker 667
- Object Pooling 667
- Customizing the Data Packets 667

What's Next? 668

Chapter 17 • Writing Database Components 669

The Data Link 669
- The *TDataLink* Class 670
- Derived Data Link Classes 670

Writing Field-Oriented Data-Aware Controls 671
- A Read-Only ProgressBar 671
- A Read-Write TrackBar 674

Creating Custom Data Links 678
- A Record Viewer Component 678

Customizing the DBGrid Component 683

Building Custom Datasets 686
- The Definition of the Classes 687
- Section I: Initialization, Opening, and Closing 690
- Section II: Movement and Bookmark Management 695
- Section III: Record Buffers and Field Management 698
- Section IV: From Buffers to Fields 702
- Testing the Stream-Based DataSet 703

A Directory in a Dataset 705
- A List as a Dataset 705
- Directory Data 706

A Dataset of Objects 710

What's Next? 713

Chapter 18 • Reporting with Rave **715**
Introducing Rave 716
  Rave: The Report Authoring Visual Environment 716
  Using the RvProject Component 719
  Rendering Formats 720
  Data Connections 721
Components of the Rave Designer 722
  Basic Components 723
  Data Access Objects 725
  Regions and Bands 726
  Data-Aware Components 728
Advanced Rave 730
  Master/Detail Reports 730
  Scripting Reports 731
  Mirroring 732
  Calculations to the Max 733
What's Next? 734

Part IV • Delphi, the Internet, and a .NET Preview **735**

Chapter 19 • Internet Programming: Sockets and Indy **737**
Building Socket Applications 737
  Foundations of Socket Programming 739
  Using Indy's TCP Components 741
  Sending Database Data over a Socket Connection 744
Sending and Receiving Mail 748
  Mail In and Out 748
Working with HTTP 750
  Grabbing HTTP Content 751
  Browsing on Your Own 756
  A Simple HTTP Server 757
Generating HTML 758
  Delphi's HTML Producer Components 759
  Producing HTML Pages 759
  Producing Pages of Data 761
  Producing HTML Tables 762
  Using Style Sheets 764
  Dynamic Pages from a Custom Server 764
What's Next? 766

Chapter 20 • Web Programming with WebBroker and WebSnap **767**
Dynamic Web Pages 768
  An Overview of CGI 768
  Using Dynamic Libraries 769
Delphi's WebBroker Technology 770
  Debugging with the Web App Debugger 772
  Building a Multipurpose WebModule 774

    Dynamic Database Reporting 776
    Queries and Forms 777
    Working with Apache 781
Practical Examples 782
    A Graphical Web Hit Counter 782
    Searching with a Web Search Engine 785
WebSnap 786
    Managing Multiple Pages 789
    Server-Side Scripts 791
    Adapters 794
    Locating Files 798
WebSnap and Databases 798
    A WebSnap Data Module 798
    The DataSetAdapter 799
    Editing the Data in a Form 801
    Master/Detail in WebSnap 803
Sessions, Users, and Permissions 805
    Using Sessions 805
    Requesting Login 807
What's Next? 808

Chapter 21 • Web Programming with IntraWeb 809

Introducing IntraWeb 810
    From Websites to Web Applications 810
    A First Look Behind the Scenes 813
    IntraWeb Architectures 815
Building IntraWeb Applications 815
    Writing Multipage Applications 818
    Sessions Management 821
    Integrating with WebBroker (and WebSnap) 823
    Controlling the Layout 824
Web Database Applications 825
    Linking to Details 827
    Moving Data to the Client Side 831
What's Next? 832

Chapter 22 • Using XML Technologies 833

Introducing XML 833
    Core XML Syntax 834
    Well-Formed XML 835
    Working with XML 836
    Managing XML Documents in Delphi 837
Programming with the DOM 838
    An XML Document in a TreeView 839
    Creating Documents Using the DOM 842
    XML Data-Binding Interfaces 846
    Using the SAX API 850
    Mapping XML with Transformations 854

XML and Internet Express 858
    The XMLBroker Component 858
    JavaScript Support 859
    Building an Example 860
Using XSLT 864
    Using XPath 864
    XSLT in Practice 865
    XSLT with WebSnap 866
    Direct XSL Transformations with the DOM 868
Processing Large XML Documents 869
    From a ClientDataSet to an XML Document 870
    From an XML Document to a ClientDataSet 871
What's Next? 873

Chapter 23 • Web Services and SOAP 875

Web Services 875
    SOAP and WSDL 876
    BabelFish Translations 876
Building Web Services 879
    A Currency Conversion Web Service 880
    Asking for Database Data 883
    Debugging the SOAP Headers 887
    Exposing an Existing Class as a Web Service 888
DataSnap over SOAP 889
    Building the DataSnap SOAP Server 889
    Building the DataSnap SOAP Client 891
    SOAP versus Other DataSnap Connections 891
Handling Attachments 892
Supporting UDDI 894
    What Is UDDI? 894
    UDDI in Delphi 7 895
What's Next? 898

Chapter 24 • The Microsoft .NET Architecture from the Delphi Perspective . . . 899

Installing the Delphi for .NET Preview 900
    Testing the Installation 901
Microsoft's .NET Platform 903
    The Common Language Infrastructure (CLI) 903
    The Common Language Runtime (CLR) 905
    Assemblies 906
The Intermediate Language 907
    Managed and Safe Code 908
    The Common Type System 909
Garbage Collection 911
    Garbage Collection and Efficiency 915
Deployment and Versioning 916
What's Next? 918

Chapter 25 • Delphi for .NET Preview: The Language and the RTL 919
Deprecated Delphi Language Features 920
    Deprecated Types 920
    Strings and Other Types 920
    Deprecated Code Features 921
New Delphi Language Features 922
    Unit Namespaces 922
    Extended Identifiers 924
    The *final* and *sealed* Keywords 924
    New Visibility and Access Specifiers 925
    Class Static Members 925
    Nested Types 926
    Multicast Events 926
    Custom Attributes 927
    Class Helpers 929
The Run-time Library and VCL 930
    Class Helpers for the RTL 930
The VCL 931
    Looking into the VCL.NET Source 931
    Further VCL Examples 933
Using Microsoft Libraries 934
ASP.NET with the Delphi Language 940
What's Next? 943

Appendix A • Extra Delphi Tools by the Author 945
CanTools Wizards 945
VclToClx Conversion Program 947
Object Debugger 948
Memory Snap 948
Licensing and Contributions 949

Appendix B • Extra Delphi Tools from Other Sources 951
Delphi Preinstalled Open-Source Components 951
Other Open-Source Projects 951
    Project JEDI 952
    GExperts 952
    The Delphree Site 952
    DUnit 952

Appendix C • Free Companion Books on Delphi 953
Essential Pascal 953
Essential Delphi 954
Delphi Power Book 954

Index 955

Introduction

THE FIRST TIME ZACK URLOCKER showed me a yet-to-be-released product code-named Delphi, I realized that it would change my work—and the work of many other software developers. I used to struggle with C++ libraries for Windows, and Delphi was and still is the best combination of object-oriented programming and visual programming not only for this operating system but also for Linux and soon for .NET.

Delphi 7 simply builds on this tradition and on the solid foundations of the VCL to deliver another astonishing and all-encompassing software development tool. Looking for database, client/server, multitier, intranet, or Internet solutions? Looking for control and power? Looking for fast productivity? With Delphi and the plethora of techniques and tips presented in this book, you'll be able to accomplish all this.

Seven Versions and Counting

Some of the original Delphi features that attracted me were its form-based and object-oriented approach, its extremely fast compiler, its great database support, its close integration with Windows programming, and its component technology. But the most important element was the Object Pascal language, which is the foundation of everything else.

Delphi 2 was even better! Among its most important additions were these: the Multi-Record Object and the improved database grid, OLE Automation support and the variant data type, full Windows 95 support and integration, the long string data type, and Visual Form Inheritance. Delphi 3 added to this the code insight technology, DLL debugging support, component templates, the TeeChart, the Decision Cube, the WebBroker technology, component packages, ActiveForms, and an astonishing integration with COM, thanks to interfaces.

Delphi 4 gave us the AppBrowser editor, new Windows 98 features, improved OLE and COM support, extended database components, and many additions to the core VCL classes, including support for docking, constraining, and anchoring controls. Delphi 5 added to the picture many more improvements of the IDE (too many to list here), extended database support (with specific ADO and InterBase datasets), an improved version of MIDAS with Internet support, the TeamSource version-control tool, translation capabilities, the concept of frames, and new components.

Delphi 6 added to all these features support for cross-platform development with the Component Library for Cross-Platform (CLX), an extended run-time library, the dbExpress database engine, Web services and exceptional XML support, a powerful Web development framework, more IDE enhancements, and a plethora of components and classes, still covered in detail in the following pages.

Delphi 7 did make some of these newer technologies more robust with improvement and fixes (SOAP support and DataSnap come to mind) and offers support for newer technologies (like Windows XP themes or UDDI), but it most importantly makes readily available an interesting set of third-party tools: the RAVE reporting engine, the IntraWeb web application development technology, and the ModelMaker design environment. Finally, it opens up a brand new world by providing (even if in a preview version) the first Borland compiler for the Pascal/Delphi language not targeting the Intel CPU, but rather the .NET CIL platform.

Delphi is a great tool, but it is also a complex programming environment that involves many elements. This book will help you master Delphi programming, including the Delphi language, components (both using the existing ones and developing your own), database and client/server support, the key elements of Windows and COM programming, and Internet and Web development.

You do not need in-depth knowledge of any of these topics to read this book, but you do need to know the basics of programming. Having some familiarity with Delphi will help you considerably, particularly after the introductory chapters. The book starts covering its topics in depth immediately; much of the introductory material from previous editions has been removed. Some of this material and an introduction to Pascal is available on my website, as discussed in Appendix C.

The Structure of the Book

The book is divided into four parts:

- Part I, "Foundations," introduces new features of the Delphi 7 Integrated Development Environment (IDE) in Chapter 1, then moves to the Delphi language and to the run-time library (RTL) and Visual Component Library (VCL). Four chapters in this part provide both foundations and coverage of the most commonly used controls, the development of advanced user interfaces, and the use of forms.
- Part II, "Delphi Object-Oriented Architectures," covers the architecture of Delphi applications, the development of custom components, the use of libraries and packages, modeling with ModelMaker, and COM+.
- Part III, "Delphi Database-Oriented Architectures," covers plain database access, in-depth coverage of the data-aware controls, client/server programming, dbExpress, InterBase, ADO, DataSnap, the development of custom data-aware controls and data sets, and reporting.
- Part IV, "Delphi, the Internet, and a .NET Preview," first discusses TCP/IP sockets, Internet protocols and Indy, then moves on to specific areas like web server-side extensions (with WebBroker, WebSnap, and IntraWeb), and finishes with XML and the development of web services.

As this brief summary suggests, the book covers topics of interest to Delphi users at nearly all levels of programming expertise, from "advanced beginners" to component developers.

In this book, I've tried to skip reference material almost completely and focus instead on techniques for using Delphi effectively. Because Delphi provides extensive online documentation, to include lists of methods and properties of components in the book would not only be superfluous, it would also make it obsolete as soon as the software changes slightly. I suggest that you read this book with the Delphi Help files at hand, to have reference material readily available.

However, I've done my best to allow you to read the book away from a computer if you prefer. Screen images and the key portions of the listings should help in this direction. The book uses just a few conventions to make it more readable. All the source code elements, such as keywords, properties, classes, and functions, appear in `this font`, and code excerpts are formatted as they appear in the Delphi editor, with boldfaced keywords and italic comments.

Free Source Code on the Web

This book focuses on examples. After the presentation of each concept or Delphi component, you'll find a working program example (sometimes more than one) that demonstrates how the feature can be used. All told, there are over 300 examples presented in the book. These programs are available in a single ZIP file of less than 2MB on both Sybex's website (`www.sybex.com`) and my website (`www.marcocantu.com`). Most of the examples are quite simple and focus on a single feature. More complex examples are often built step-by-step, with intermediate steps including partial solutions and incremental improvements.

NOTE *Some of the database examples also require you to have the Delphi sample database files installed; they are part of the default Delphi installation. Others require the InterBase EMPLOYEE sample database (and also the InterBase server, of course).*

On my website there is also an HTML version of the source code, with full syntax highlighting, along with a complete cross-reference of keywords and identifiers (class, function, method, and property names, among others). The cross-reference is an HTML file, so you'll be able to use your browser to easily find all the programs that use a Delphi keyword or identifier you're looking for (not a full search engine, but close enough).

The directory structure of the sample code is quite simple. Basically, each chapter of the book has its own folder, with a subfolder for each example (e.g., `03\FilesList`). In the text, the examples are simply referenced by name (e.g., FilesList).

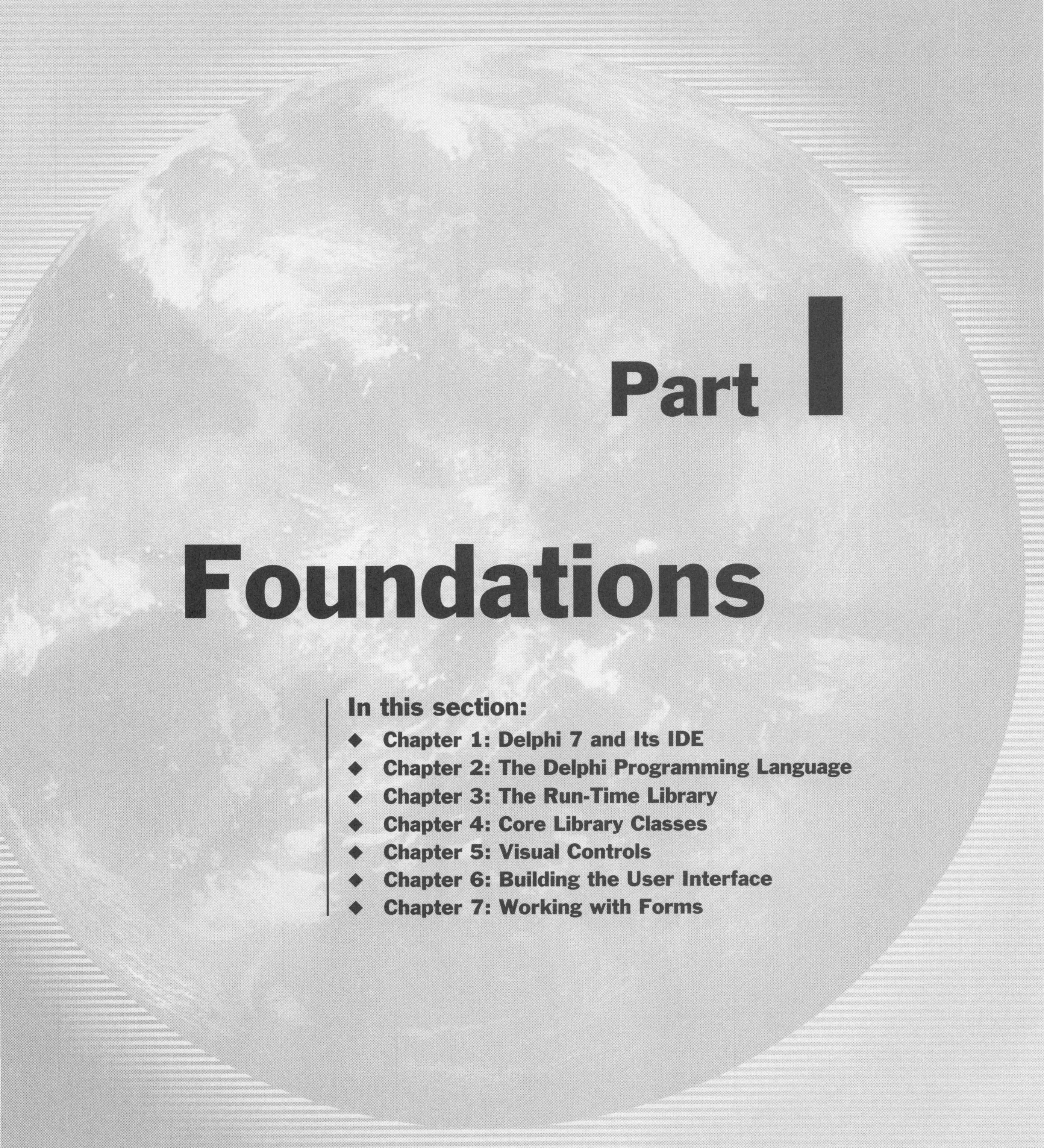
NOTE *Be sure to read the source code archive's readme file, which contains important information about using the software legally and effectively.*

How to Reach the Author

If you find any problems in the text or examples in this book, both the publisher and I would be happy to hear from you. Besides reporting errors and problems, please give us your unbiased opinion of the book and tell us which examples you found most useful and which you liked least. There are several ways you can provide this feedback:

- On the Sybex website (`www.sybex.com`), you'll find updates to the text or code as necessary. To comment on this book, click the Contact Sybex link and then choose Book Content Issues. This link displays a form where you can enter your comments.
- My own website (`www.marcocantu.com`) hosts further information about the book and about Delphi, where you might find answers to your questions. The site has news and tips, technical articles, free online books (outlined in Appendix C), white papers, Delphi links, and my collection of Delphi components and tools (covered in Appendix A).

- I have also set up a newsgroup section specifically devoted to my books and to general Delphi Q&A. Refer to my website for a list of the newsgroup areas and for the instructions to subscribe to them. (In fact, these newsgroups are totally free but require a login password.) The newsgroups can also be accessed via a Web interface through a link you can find on my site.
- Finally, you can reach me via e-mail at `marco@marcocantu.com`. For technical questions, please try using the newsgroups first, as you might get answers earlier and from multiple people. My mailbox is usually overflowing and, regretfully, I cannot reply promptly to every request. (Please write to me in English or Italian.)



Part I

Foundations

In this section:

- **Chapter 1: Delphi 7 and Its IDE**
- **Chapter 2: The Delphi Programming Language**
- **Chapter 3: The Run-Time Library**
- **Chapter 4: Core Library Classes**
- **Chapter 5: Visual Controls**
- **Chapter 6: Building the User Interface**
- **Chapter 7: Working with Forms**

Chapter 1

Delphi 7 and Its IDE

In a visual programming tool such as Delphi, the role of the integrated development environment (IDE) is at times even more important than the programming language. Delphi 7 provides some interesting new features on top of the rich IDE of Delphi 6. This chapter examines these new features, as well as features added in other recent versions of Delphi. We'll also discuss a few traditional Delphi features that are not well known or obvious to newcomers. This chapter isn't a complete tutorial of the IDE, which would require far too much space; it's primarily a collection of tips and suggestions aimed at the average Delphi user.

If you *are* a beginning programmer, don't be afraid. The Delphi IDE is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi Companion Tools CD) with a tutorial that introduces the development of Delphi applications. You can find a simpler introduction to Delphi and its IDE in my *Essential Delphi* online book (discussed in Appendix C, "Free Companion Books on Delphi"). Throughout this book, I'll assume you already know how to carry out the basic hands-on operations of the IDE; all the chapters after this one focus on programming issues and techniques.

This chapter covers the following topics:

- Moving around the IDE
- The editor
- The Code Insight technology
- Designing forms
- The Project Manager
- Delphi files

Editions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

- The "Personal" edition is aimed at Delphi newcomers and casual programmers and has support for neither database programming nor any of the other advanced features of Delphi.
- The "Professional Studio" edition is aimed at professional developers. It includes all the basic features, plus database programming support (including ADO support), basic web server support (WebBroker), and some of the external tools, including ModelMaker and IntraWeb. This book generally assumes you are working with at least the Professional edition.
- The "Enterprise Studio" edition is aimed at developers building enterprise applications. It includes all the XML and advanced web services technologies, CORBA support, internationalization, three-tier architecture, and many other tools. Some chapters of this book cover features included only in Delphi Enterprise; these sections are specifically identified.
- The "Architect Studio" edition adds to the Enterprise edition support for Bold, an environment for building applications that are driven at run time by a UML model and capable of mapping their objects both to a database and to the user interface, thanks to a plethora of advanced components. Bold support is not covered in this book.

Besides the different editions available, there are ways to customize the Delphi environment. In the screen illustrations throughout the book, I've tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which may be reflected in some of the screen shots.

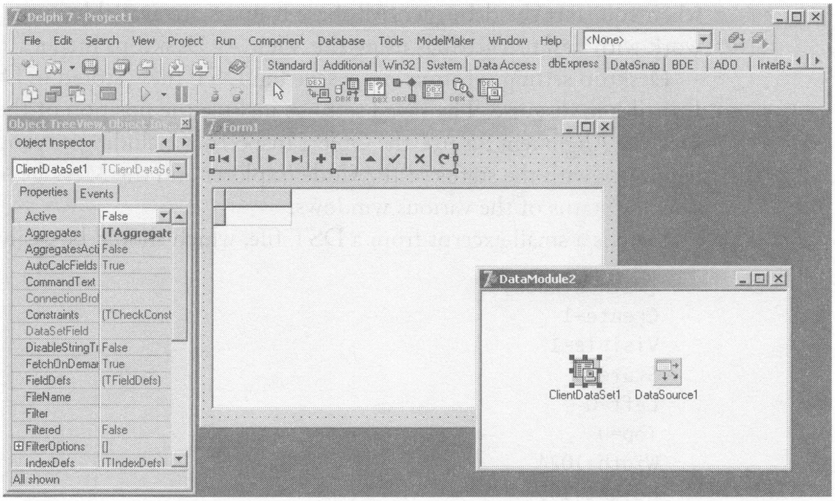
The Professional and higher versions of Delphi 7 include a working copy of Kylix 3, in the Delphi language edition. Other than references to the CLX library and cross-platform features of Delphi, this book doesn't cover Kylix and Linux development. You can refer to *Mastering Kylix 2* (Sybex, 2002) for more information on the topic. (There aren't many differences between Kylix 2 and Kylix 3 in the Delphi language version. The most important new feature of Kylix 3 is its support of the C++ language.)

An Overview of the IDE

When you work with a visual development environment, your time is spent in two different portions of the application: visual designers and the code editor. Designers let you work with components at the visual level (such as when you place a button on a form) or at a non-visual level (such as when you place a DataSet component on a data module). You can see a form and a data module in action in Figure 1.1. In both cases, designers allow you to choose the components you need and set the initial value of the components' properties.

The code editor is where you write code. The most obvious way to write code in a visual environment involves responding to events, beginning with events attached to operations performed by program users, such as clicking on a button or selecting an item of a list box. You can use the same approach to handle internal events, such as events involving database changes or notifications from the operating system.

As programmers become more knowledgeable about Delphi they often begin by writing mainly event-handling code and then move to writing their own classes and components, and often end up spending most of their time in the editor. Because this book covers more than visual programming, and tries to help you master the entire power of Delphi, as the text proceeds you'll see more code and fewer forms.



Delphi 7 - Project1
File Edit Search View Project Run Component Database Tools ModelMaker Window Help
<None>
Standard Additional Win32 System Data Access dbExpress DataSnap BDE ADO InterBa
Object TreeView, Object Ins
Object Inspector
ClientDataSet1 TClientDataSe
Properties Events
All shown
Form1
DataModule2
ClientDataSet1
DataSource1

Active	False
Aggregates	(TAggregate
AggregatesActi	False
AutoCalcFields	True
CommandText	
ConnectionBrol	
Constraints	(TCheckConst
DataSetField	
DisableStringTr	False
FetchOnDemar	True
FieldDefs	(TFieldDefs)
FileName	
Filter	
Filtered	False
⊞FilterOptions	[]
IndexDefs	(TIndexDefs)

FIGURE 1.1
A form and a data module in the Delphi 7 IDE

An IDE for Two Libraries

An important change appeared for the first time in Delphi 6. The IDE now lets you work on two different visual libraries: VCL (Visual Component Library) and CLX (Component Library for Cross-Platform). When you create a new project, you simply choose which of the two libraries you want to use, starting with the File ➢ New ➢ Application command for a classic VCL-based Windows program and with the File ➢ New ➢ CLX Application command for a new CLX-based portable application.

NOTE *CLX is Delphi's Cross Platform library, which allows you to recompile your code with Kylix to run under Linux. You can read more about VCL versus CLX in Chapter 5, "Visual Controls." Using CLX is even more interesting in Delphi 7, because the Delphi language version of Kylix ships with the Windows product.*

When you create a new project or open an existing one, the Component Palette is arranged to show only the controls related to the current library (although most of the controls are shared). When you work with a non-visual designer (such as a data module), the tabs of the Component Palette that host only visual components are hidden from view.

Desktop Settings

Programmers can customize the Delphi IDE in various ways—typically, opening many windows, arranging them, and docking them to each other. However, you'll often need to open one set of windows at design time and a different set at debug time. Similarly, you might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

For this reason, Delphi lets you save a given arrangement of IDE windows (called a *desktop*, or a Global Desktop, to differentiate from a Project Desktop) with a name and restore it easily. You can also make one of these groupings your default debugging setting, so that it will be restored automatically when you start the debugger. All these features are available in the Desktops toolbar. You can also work with desktop settings using the View ➢ Desktops menu.

Desktop setting information is saved in DST files (stored in Delphi's `bin` directory), which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78

[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1

[AlignmentPalette]
Create=1
Visible=0
...
```

Desktop settings override project settings, which are saved in a DSK file with a similar structure. Desktop settings help eliminate problems that can occur when you move a project between machines (or between developers) and have to rearrange the windows to your liking. Delphi separates per-user global desktop settings and per-project desktop settings, to better support team development.

TIP *If you open Delphi and cannot see the form or other windows, I suggest you try checking (or deleting) the desktop settings (from Delphi's* `bin` *directory). If you open a project received by a different user and cannot see some of the windows or dislike the desktop layout, reload your global desktop settings or delete the project DSK file.*

Environment Options

Quite a few recent updates relate to the commonly used Environment Options dialog box. The pages of this dialog box were rearranged in Delphi 6, moving the Form Designer options from the Preferences page to the new Designer page. In Delphi 6 there were also a few new options and pages:

- The Preferences page of the Environment Options dialog box has a check box that prevents Delphi windows from automatically docking with each other.

- The Environment Variables page allows you to see system environment variables (such as the standard pathnames and OS settings) and set user-defined variables. The nice point is that you can use both system- and user-defined environment variables in each of the dialog boxes of the IDE—for example, you can avoid hard-coding commonly used pathnames, replacing them with a variable. In other words, the environment variables work similarly to the `$DELPHI` variable, referring to Delphi's base directory, but can be defined by the user.
- In the Internet page you can choose the default file extensions used for HTML and XML files (mainly by the WebSnap framework) and also associate an external editor with each extension.

About Menus

The main Delphi menu bar (which in Delphi 7 has a more modern look) is an important way to interact with the IDE, although you'll probably accomplish most tasks using shortcut keys and shortcut menus. The menu bar doesn't change much in reaction to your current operations: You need to click the right mouse button for a full list of the operations you can perform on the current window or component.

The menu bar can change considerably depending on third-party tools and wizards you've installed. In Delphi 7, ModelMaker has its own menu. You'll see other menus by installing popular add-ons like GExperts or even my own wizards (see Appendix B, "Extra Delphi Tools from other Sources" and A, "Extra Delphi Tools by the Author," respectively, for more details).

A relevant menu added to Delphi in recent editions is the Window menu in the IDE. This menu lists the open windows; previously, you could obtain this list using the Alt+0 key combination or the View ➢ Window List menu item. The Window menu is really handy, because windows often end up behind others and are hard to find. You can control the alphabetic sort order of this menu using a setting in the Windows Registry: Look for the `Main Window` subkey of Delphi (under `HKEY_CURRENT_USER\Software\Borland\Delphi\7.0`). This Registry key uses a string (in place of Boolean values), where '-1' and 'True' indicate true and '0' and 'False' indicate false.

TIP *In Delphi 7, the Window menu ends with a new command: Next Window. This command is particularly useful in the form of a shortcut, Alt+End. Jumping around the various windows of the IDE has never been so simple (at least, without add-on tools).*

The Environment Options Dialog Box

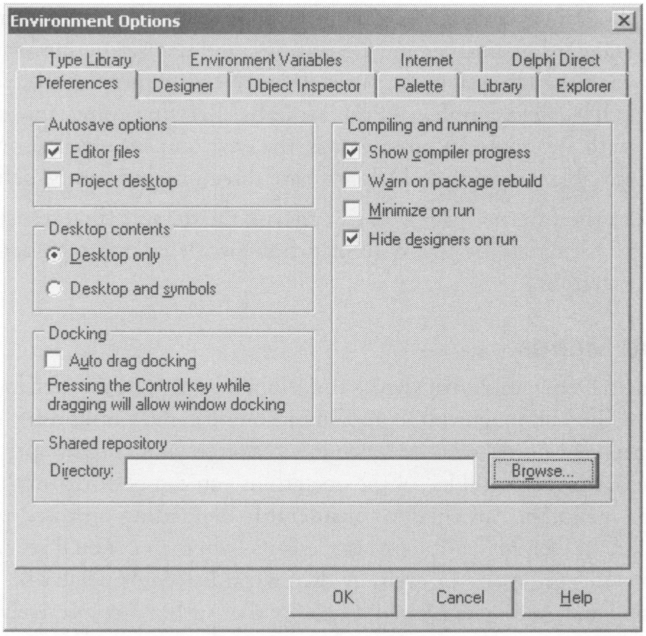
As I've mentioned, some of the IDE settings require you to edit the Registry directly. I'll discuss a few more of these settings in this chapter. Of course, the most common settings can be easily tuned using the Environment Options dialog box, which is available in the Tools menu along with the Editor Options and the Debugger Options. Most of the settings are quite intuitive and well described in the Delphi Help file. Figure 1.2 shows my standard settings for the Preferences page of this dialog box.

The To-Do List

Another feature added in Delphi 5 but still quite underused is the to-do list. This is a list of tasks you still have to do to complete a project—it's a collection of notes for the programmer (or programmers; this tool can be very handy in a team). Although the idea is not new, the key concept of the to-do list in Delphi is that it works as a two-way tool.

FIGURE 1.2

The Preferences page of the Environment Options dialog box


Environment Options
Type Library
Environment Variables
Internet
Delphi Direct
Preferences
Designer
Object Inspector
Palette
Library
Explorer
Browse...
OK
Cancel
Help

Autosave options
- ☑ Editor files
- ☐ Project desktop

Desktop contents
- ● Desktop only
- ○ Desktop and symbols

Docking
- ☐ Auto drag docking

Pressing the Control key while dragging will allow window docking

Compiling and running
- ☑ Show compiler progress
- ☐ Warn on package rebuild
- ☐ Minimize on run
- ☑ Hide designers on run

Shared repository

Directory:

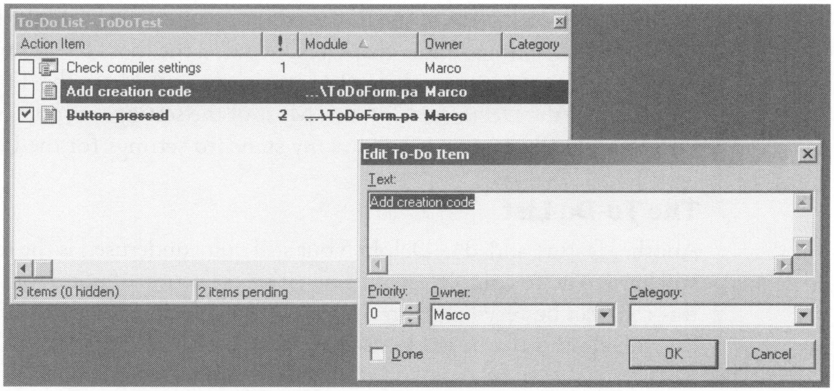
You can add or modify to-do items by adding special `TODO` comments to the source code of any file of a project; you'll then see the corresponding entries in the list. In addition, you can visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look in the source code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // TODO -oMarco: Add creation code
end;
```

The same item can be visually edited in the window shown in Figure 1.3, along with the To-Do List window.

FIGURE 1.3

The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code.


To-Do List - ToDoTest
Edit To-Do Item
OK
Cancel

Action Item	!	Module	Owner	Category
☐ Check compiler settings	1		Marco	
☐ **Add creation code**		**...\ToDoForm.pa**	**Marco**	
☑ ~~**Button pressed**~~	~~**2**~~	~~**...\ToDoForm.pa**~~	~~**Marco**~~	

3 items (0 hidden)

2 items pending

Text: Add creation code

Priority: 0

Owner: Marco

Category:

☐ Done

The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the Ctrl+A key combination in the To-Do List window or right-click in the window and select Add from the shortcut menu. These items are saved in a special file with the same root name as the project file and a .TODO extension.

You can use multiple options with a `TODO` comment. You can use `-o` (as in the previous code excerpt) to indicate the owner (the programmer who entered the comment), the `-c` option to indicate a category, or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

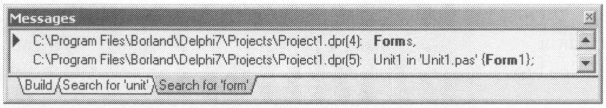
Delphi treats everything after the colon—up to the end of the line or the closing brace, depending on the type of comment—as the text of the to-do item.

Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from `TODO` to `DONE`. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, which can automatically collect to-do information from the source code files as you type them, sort and filter them, and export them to the Clipboard as plain text or an HTML table. All these options are available on the context menu.

Extended Compiler Messages and Search Results in Delphi 7

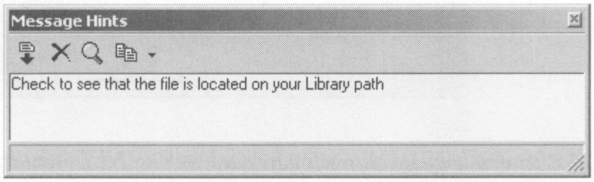
A small Messages window appears by default below the editor; it displays both compiler messages and search results. This window has been considerably modified in Delphi 7. First, search results are displayed in a different tab so they do not interfere with compiler messages as they did in the past. Second, every time you do a different search you can request that Delphi show the results in a different page, so the results of previous search operations remain available:



```
C:\Program Files\Borland\Delphi7\Projects\Project1.dpr(4):  Forms,
C:\Program Files\Borland\Delphi7\Projects\Project1.dpr(5):  Unit1 in 'Unit1.pas' {Form1};
```

You can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the tabs of this window. (The same commands work for other tabbed views.)

If compiler errors occur, you can activate another new window with the command View ➢ Additional Message Info. As you compile a program, this Message Hints window will provide extra information for some common error messages, offering suggestions about how to fix them:



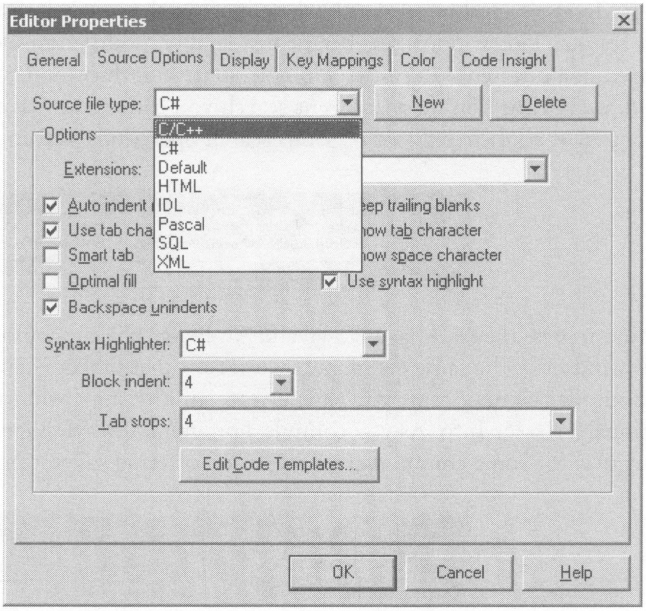
Check to see that the file is located on your Library path

This type of help is intended more for novice programmers, but it might be handy to keep this window around. It's important to realize that this information is thoroughly customizable: A project development leader can put appropriate descriptions of common errors in a form that means something specific to new developers. To do so, follow the comments in the file hosting the settings for this feature, the `msginfo70.ini` file of Delphi's `bin` folder.

The Delphi Editor

On the surface, Delphi's editor doesn't appear to have changed much for version 7 of the IDE. However, behind the scenes, it is a totally new tool. Besides using it to work on files in the Object Pascal language (or the Delphi language, as Borland prefers to call it now), you can now use it to work on other files used in Delphi development (such as SQL, XML, HTML, and XSL files), as well as files in other languages (including C++ and C#). XML and HTML editing was already available in Delphi 6, but the changes in this version are significant. For example, while editing an HTML file, you have support for both syntax highlighting and code completion.

The editor settings used on each file (including the behavior of keys like Tab) depend on the extension of the file being opened. You can configure these settings in the new Source Options page of the Editor Properties dialog box, displayed in Figure 1.4. This feature has been extended and made more open, so you can even configure the editor by providing a DTD for XML-based file formats or by writing a custom wizard that provides syntax highlighting for other programming languages. Another feature of the editor, code templates, is now language specific (your predefined Delphi templates will make little sense in HTML or C#).



Source file type: C#

Extensions:

☑ Auto indent

☑ Use tab cha

☐ Smart tab

☐ Optimal fill

☑ Backspace unindents

ep trailing blanks

how tab character

how space character

☑ Use syntax highlight

Syntax Highlighter: C#

Block indent: 4

Tab stops: 4

FIGURE 1.4
The multiple languages supported by the Delphi IDE can be associated with various file extensions in the Source Options page of the Editor Properties dialog box.

NOTE *C# is the new language Microsoft introduced with its .NET architecture. Borland is expected to support C# in its own .NET environment, currently code-named Galileo.*

Considering only the Delphi language, the editor included in the IDE hasn't changed much in recent versions. However, it has a few features that many Delphi programmers don't know about and use, so I think it's worth a brief examination.

The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Ctrl+Shift+Tab to move in the opposite direction). You can drag-and-drop the tabs with the unit names in the upper portion of the editor to change their order, so that you can use a single Ctrl+Tab to move between the units you are working on any given time. The editor's shortcut menu has also a Pages command, which lists all the available pages in a submenu (a handy feature when many units are loaded).

You can also open multiple editor windows, each hosting multiple tabs. Doing so is the only way to see the source code of two units alongside each other. (Actually, when I need to compare two Delphi units, I invariably use Beyond Compare—`www.scootersoftware.com`—a superb, low-cost file comparison utility written in Delphi.)

Several options affect the editor, as you can see in the Editor Properties dialog box in Figure 1.4. However, you have to go to the Preferences page of the Environment Options dialog box (see Figure 1.2) to set the editor's AutoSave feature. This option forces the editor to save all of your source code files each time you run the program, preventing data loss in the (rare) case the program crashes badly in the debugger.

Delphi's editor provides many commands, including some that date back to its WordStar emulation ancestry (of the early Turbo Pascal compilers). I won't discuss the various settings of the editor, because they are quite intuitive and are described in the online help. Notice, though, that the page of the help describing the keyboard shortcuts is accessible as a whole only if you look up the *shortcuts* index item.

TIP *A tip to remember is that using the Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. In addition, you can copy text instead of moving it, by pressing the Ctrl key while dragging.*

The Code Explorer

The Code Explorer window, which is generally docked on the side of the editor, lists all the types, variables, and routines defined in a unit, plus other units appearing in `uses` statements. For complex types, such as classes, the Code Explorer can list detailed information, including a list of fields, properties, and methods. All the information is updated as soon as you begin typing in the editor.

You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration. You can also modify variables, properties, and method names directly in the Code Explorer. However, as you'll see, if you want a visual tool to use when you work on your classes, ModelMaker provides many more features.

Although all this functionality is obvious after you've used Delphi for a few minutes, some features of the Code Explorer are not so intuitive. You have full control of the information layout. And, you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer (collapsing the tree can help you make your selections more quickly). You can configure the Code Explorer by using the corresponding page of the Environment Options, as shown in Figure 1.5.

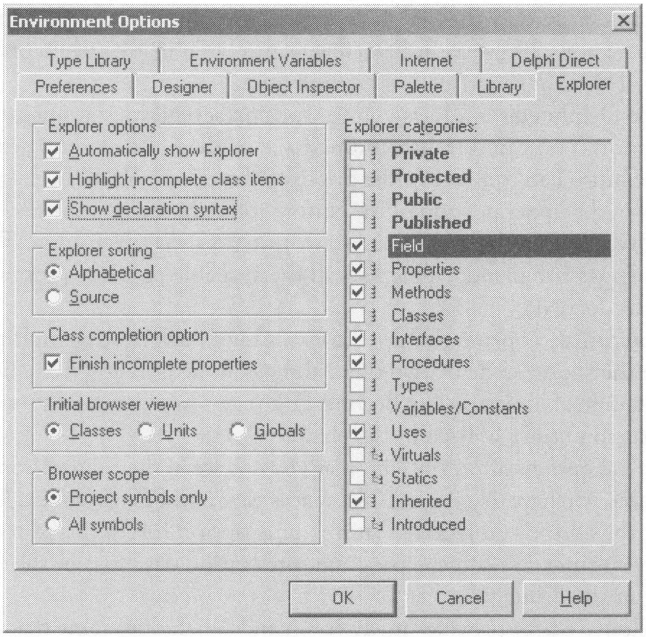


FIGURE 1.5 You can configure the Code Explorer in the Environment Options dialog box.

Explorer options
☑ Automatically show Explorer
☑ Highlight incomplete class items
☑ Show declaration syntax

Explorer sorting
● Alphabetical
○ Source

Class completion option
☑ Finish incomplete properties

Initial browser view
● Classes ○ Units ○ Globals

Browser scope
● Project symbols only
○ All symbols

Explorer categories:
☐ **Private**
☐ **Protected**
☐ **Public**
☐ **Published**
☑ Field
☑ Properties
☑ Methods
☐ Classes
☑ Interfaces
☑ Procedures
☐ Types
☐ Variables/Constants
☑ Uses
☐ Virtuals
☐ Statics
☑ Inherited
☐ Introduced

Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view—it simply adds the node in the tree. For example, if you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer; on the contrary, the used units are listed as main nodes instead of being kept in the `Uses` folder. I generally disable the Types, Classes, and Variables/Constants selections.

Because each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, because this arrangement requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer provides a handy way of navigating the source code of a large unit—when you double-click a method in the Code Explorer, the focus moves to the definition in the class declaration. You can use Module Navigation (the Ctrl+Shift combination with the Up or Down arrow key) to jump from the definition of a method or procedure to its complete definition (or back again).

NOTE *Some of the Explorer Categories shown in Figure 1.5 are used by the Project Browser, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.*

Browsing in the Editor

Another feature of the editor is *Tooltip symbol insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the library.

WARNING *Although it may seem like a good idea at first, you cannot use Tooltip symbol insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, the Tooltip won't appear.*

The real bonus of this feature, however, is that you can turn it into a navigational aid called *code browsing*. When you hold down the Ctrl key and move the mouse over the identifier, Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of links in web browsers, and the pointer changes to a hand whenever it's positioned on the link.

For example, you can Ctrl+click the `TLabel` identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again, as in a web browser—using the Browse Back and Browse Forward buttons in the top-right corner of the editor windows or the keystrokes Alt+Left arrow or Alt+Right arrow. You can also click the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

How can you jump directly to the VCL source code if it is not part of your project? The editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment.

Class Completion

Delphi's editor can also help by generating some source code for you, completing what you've already written. This feature is called *class completion*, and you activate it by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, because Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi simplify life in a similar way for programmers who write a little extra code behind event handlers. This code-generation feature applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration

```
public
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }
procedure TForm1.Hello(MessageText: string);
begin
end;
```

This feature is really handy compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin...end` code for every method copied. Class completion also works the other way around: You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

The most important and useful example of class completion is the automatic generation of code to support properties declared in classes. For example, if you type in a class

```
property Value: Integer;
```

and press Ctrl+Shift+C, Delphi will turn the line into

```
property Value: Integer read fValue write SetValue;
```

Delphi will also add the `SetValue` method to the class declaration and provide a default implementation for it. You'll find more on properties in the next chapter.

Code Insight

In addition to the Code Explorer, class completion, and the navigational features, the Delphi editor supports the *code insight* technology. Collectively, the code insight techniques are based on a constant background parsing of both the source code you write and the source code of the system units your source code refers to.

Code insight comprises five capabilities: code completion, code templates, code parameters, Tooltip expression evaluation, and Tooltip symbol insight. This last feature was already covered in the section "Browsing in the Editor"; the other four are discussed in the following subsections. You can enable, disable, and configure each of these features in the Code Insight page of the Editor Properties dialog box.

CODE COMPLETION

Code completion allows you to choose the property or method of an object simply by looking it up on a list or by typing its initial letters. To activate this list, you just type the name of an object, such as `Button1`, then add the dot, and wait. To force the display of the list, press Ctrl+spacebar; to remove it when you don't want it, press Esc. Code completion also lets you look for a proper value in an assignment statement.

As you begin typing, the list filters its content according to the initial portion of the element you've inserted. The code completion list uses colors and shows more details to help you distinguish different items. In Delphi, you can customize these colors in the Code Insight page of the Editor Options dialog box. Another feature is that in the case of functions with parameters, parentheses are included in the generated code, and the parameters list hint is displayed immediately.

As you type `:=` after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click it to change the order of the items, sorting either by scope or by name; you can also resize the window.

Since Delphi 6, code completion also works in the interface section of a unit. If you press Ctrl+spacebar while the cursor is inside the class definition, you'll get a list of virtual methods you can override (including abstract methods), the methods of implemented interfaces, the base class properties, and eventually system messages you can handle. Simply selecting one of them will add the proper method to the class declaration. In this particular case, the code completion list allows multiple selection.

TIP *When the code you've written is incorrect, code insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific code insight errors in the Message pane (which must already be open—it doesn't open automatically to display compilation errors). To activate this feature, you need to set an undocumented Registry entry, setting the string key* `\Delphi\7.0\Compiling\ShowCodeInsiteErrors` *to the value '1'.*

Code completion includes some advanced features that aren't easy to spot. One that I find particularly useful relates to the discovery of symbols in units not used by your project. As you invoke it (with Ctrl+spacebar) over a blank line, the list also includes symbols from common units (such as Math, StrUtils, and DateUtils) not already included in the `uses` statement of the current unit. By selecting one of these *external* symbols, Delphi adds the unit to the `uses` statement for you. This feature (which doesn't work inside expressions) is driven by a customizable list of extra units, stored in the Registry key `\Delphi\7.0\CodeCompletion\ExtraUnits`.

TIP *Delphi 7 adds the ability to browse to the declaration of items in the code completion list by Ctrl+clicking on any identifier in the list.*

CODE TEMPLATES

This feature lets you insert one of the predefined code templates, such as a complex statement with an inner `begin...end` block. Code templates must be activated manually, by pressing Ctrl+J to show a list of all of the templates. If you type a few letters (such as a keyword) before pressing Ctrl+J, Delphi will list only the templates starting with those letters.

You can add custom code templates, so that you can build your own shortcuts for commonly used blocks of code. For example, if you use the `MessageDlg` function often, you might want to add a template for it. To modify templates, go to the Source Options page of the Editor Options dialog box, select Pascal from the Source File Type list, and click the Edit Code Templates button. Doing so opens the new Delphi 7 Code Templates dialog box. At this point, click the Add button, type in a new template name (for example, **mess**), type a description, and then add the following text to the template body in the Code memo control:

```
MessageDlg ('|', mtInformation, [mbOK], 0);
```

Now, every time you need to create a message dialog box, you simply type **mess** and then press Ctrl+J, and you get the full text. The vertical line (or pipe) character indicates the position within the source code where the cursor will be in the editor after you expand the template. You should choose the position where you want to begin typing to complete the code generated by the template.

Although code templates might seem at first to correspond to language keywords, they are a more general mechanism. They are saved in the `DELPHI32.DCI` file, a text file in a rather simple format that you can edit directly. Delphi 7 also allows you to export the settings for a language to a file and import them, making it easier for developers to exchange their own customized templates.

CODE PARAMETERS

While you are typing a function or method, code parameters display the data type of the function's or method's parameters in a hint or Tooltip window. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a pop-up hint window. To force the display of code parameters, you can press Ctrl+Shift+spacebar. As a further help, the current parameter appears in bold type.

TOOLTIP EXPRESSION EVALUATION

Tooltip expression evaluation is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor. In the case of an expression, you typically need to select it in the editor and then move the mouse over the highlighted text.

More Editor Shortcut Keys

The editor has many more shortcut keys that depend on the editor style you've selected. Here are a few of the lesser-known shortcuts:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a "gutter" margin on the side of the editor. To jump back to the bookmark, press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the facts that a new bookmark can override an existing one and that bookmarks are not persistent (they are lost when you close the file).
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is set by the Block Indent option in the Editor page of the Editor Options dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R begins recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that might have little meaning once you've moved on to a different source code file. Editor macros are quite useful for performing multistep operations over and over again, such as reformatting source code or arranging data more legibly in source code.
- While holding down the Alt key, you can drag the mouse to select rectangular areas within the editor, not just consecutive lines and words.

Loadable Views

Another important feature introduced in Delphi 6 is support for multiple views in the editor. For any single file loaded in the IDE, the editor can show multiple views, defined programmatically and added to the system, and then loaded for given files—hence the name *loadable* views.

The most frequently used view is the Diagram page, which was available in Delphi 5 data modules, although it was less powerful. Another set of views is available in web applications, including an HTML Script view, an HTML Result preview, and many others discussed in Chapters 20 ("Web Programming with WebBroker and WebSnap") and 22 ("Using XML Technologies"). You can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the bottom tabs of this editor; Ctrl+Tab changes the pages (or files) shown in the upper tabs.

THE DIAGRAM VIEW

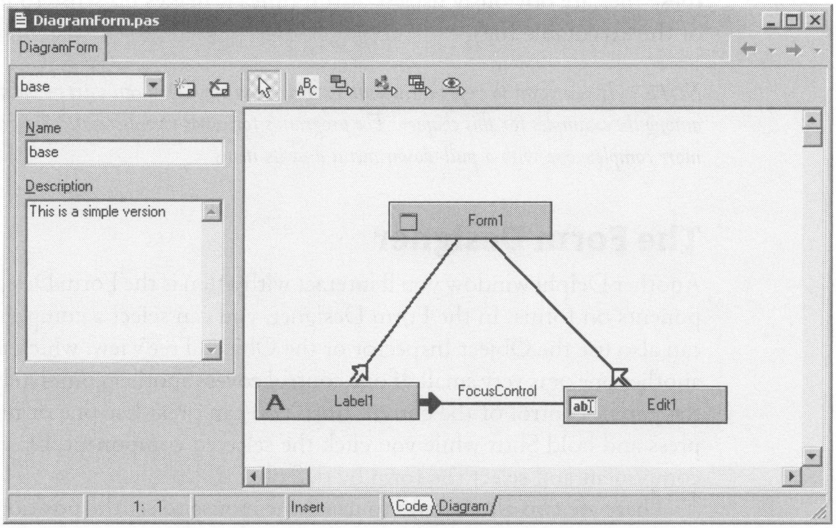
The Diagram view shows dependencies among components, including parent/child relations, ownership, linked properties, and generic relations. For dataset components, it also supports master/detail relations and lookup connections. You can even add your comments in text blocks linked to specific components.

The diagram is not built automatically. You must drag components from the Tree view to the diagram, which will automatically display the existing relations among the components you drop there. You can select multiple items from the Object TreeView and drag them all at once to the Diagram page.

What's nice is that you can set properties by simply drawing arrows between the components. For example, after moving an edit and a label to the diagram, you can select the Property Connector icon, click the label, and drag the mouse cursor over the edit. When you release the mouse button, the Diagram view will set up a property relation based on the `FocusControl` property, which is the only property of the label referring to an edit control. This situation is depicted in Figure 1.6.

FIGURE 1.6

The Diagram view shows relationships among components (and even allows you to set them up).

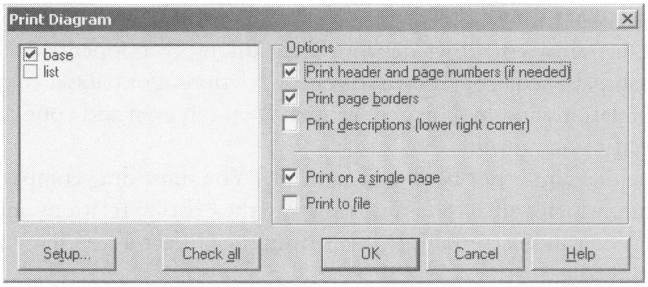


Name: base

Description: This is a simple version

As you can see, setting properties is *directional*: If you drag the property relation line from the edit to the label, you end up trying to use the label as the value of a property of the edit box. Because this isn't possible, you'll see an error message indicating the problem and offering to connect the components in the opposite way. The Diagram view allows you to create multiple diagrams for each Delphi unit—that is, for each form or data module. You give a name to the diagram and possibly add a description, click the New Diagram button, prepare another diagram, and you'll be able to switch back and forth between diagrams using the combo box available in the toolbar of the Diagram view.

Although you can use the Diagram view to set up relations, its main role is to document your design. For this reason, it is important to be able to print the content of this view. Use the standard File ➢ Print command while the Diagram view is active, and Delphi prompts you for options as shown in Figure 1.7. You can customize the output in many ways.



☑ base
☐ list

☑ Print header and page numbers (if needed)
☑ Print page borders
☐ Print descriptions (lower right corner)
☑ Print on a single page
☐ Print to file

FIGURE 1.7
The Print Options for the Diagram view

The information in the Diagram view is saved in a separate file, not as part of the DFM file. Delphi 5 used design-time information (DTI) files, which had a structure similar to INI files. Delphi 6 and 7 can still read the older .DTI format, but they use the new Delphi Diagram Portfolio format (.DDP). These files use the DFM binary format (or a similar format), so they are not editable as text. All of these files are obviously useless at run time (it makes no sense to include them in the compilation of the executable file).

NOTE *If you want to experiment with the Diagram view, you can start by opening the DiagramDemo project included among the examples for this chapter. The program's form has two associated diagrams: the one in Figure 1.6 and a much more complex one with a pull-down menu and its items.*

The Form Designer

Another Delphi window you'll interact with often is the Form Designer, a visual tool for placing components on forms. In the Form Designer, you can select a component directly with the mouse; you can also use the Object Inspector or the Object TreeView, which is handy when a control is behind another one or is very small. If one control covers another completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click the selected component. Doing so will deselect the current component and select the form by default.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position (when the Snap To Grid option is active), as is holding down Alt while using the mouse to move the control. If you press Ctrl+Shift along with an arrow key, the component will move only at grid intervals.

By pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. Again, you can also do so with the mouse and the Alt key.

To align multiple components or make them the same size, you can select them and set the `Top`, `Left`, `Width`, or `Height` property for all of them at the same time. To select several components, click them with the mouse while holding down the Shift key; or, if all the components fall into a rectangular area, drag the mouse to "draw" a rectangle surrounding them. To select child controls (say, the buttons inside a panel), drag the mouse within the panel while holding down the Ctrl key—otherwise, you move the panel. When you've selected multiple components, you can also set their relative position using the Alignment dialog box (with the Align command of the form's shortcut menu) or the Alignment Palette (accessible through the View ➢ Alignment Palette menu command).

When you've finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component. This command is particularly helpful, because Undo operations on forms are limited (you can only Undelete), but the setting is not persistent.

Among its other features, the Form Designer offers several Tooltip hints:

- As you move the pointer over a component, the hint shows you the name and type of the component. Since version 6, Delphi offers extended hints, with details about the control's position, size, tab order, and more. This is an addition to the Show Component Captions environment setting, which I keep active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the `Left` and `Top` properties).

Finally, you can save DFM (Delphi Form Module) files in the old binary resource format, instead of the plain text format, which is the default. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the Designer page of the Environment Options dialog box. In the same page, you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box).

Having DFM files stored as text lets you operate more effectively with version-control systems. Programmers won't get a real advantage from this feature, because you could already open the binary DFM files in the Delphi editor with a specific command from the designer's shortcut menu. Version-control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file.

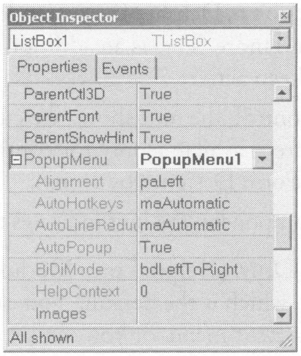
In any case, if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFMs are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded more quickly).

NOTE *Text DFM files are more portable between versions of Delphi than their binary version. Although an older version of Delphi might not accept a new property of a control in a DFM created by a newer version of Delphi, the older Delphis will still be able to read the rest of the text DFM file. If the newer version of Delphi adds a new data type, though, older Delphis will be unable to read the newer Delphi's binary DFMs at all. Even if this doesn't sound likely, remember that 64-bit operating systems are just around the corner. When in doubt, save in text DFM format. Also note that all versions of Delphi support text DFMs, using the command-line tool Convert in the* `bin` *directory. Finally, keep in mind that the CLX library uses the XFM extension instead of the DFM extension, both in Delphi and Kylix.*

The Object Inspector

To see and change properties of components placed on a form (or another designer) at design time, you use the Object Inspector. Compared to the early versions of Delphi the Object Inspector has a number of new features. The latest, introduced in Delphi 7, is the use of a bold font to highlight properties that have a value different from the default.

Another important change (introduced in Delphi 6) is the ability of the Object Inspector to expand component references in place. Properties referring to other components are displayed in a different color and can be expanded by selecting the + symbol on the left, as is the case with an internal subcomponent. You can then modify the properties of that other component without having to select it. Here you can see a connected component (a pop-up menu) expanded in the Object Inspector while working on another component (a list box):



ParentCtl3D	True
ParentFont	True
ParentShowHint	True
⊟PopupMenu	**PopupMenu1**
Alignment	paLeft
AutoHotkeys	maAutomatic
AutoLineRedu	maAutomatic
AutoPopup	True
BiDiMode	bdLeftToRight
HelpContext	0
Images	

This interface-expansion feature also supports subcomponents, as demonstrated by the new LabeledEdit control. A related feature of the Object Inspector lets you select the component referenced by a property. To do this, double-click the property value with the left mouse button while pressing the Ctrl key. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the Menu property of the form and Ctrl+double-clicking the value of this property. Doing so selects the main menu indicated as the value of the property in the Object Inspector.

Here are some other recent changes of the Object Inspector:

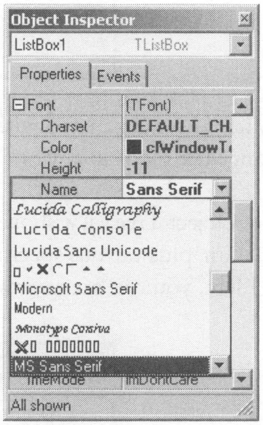
- The list at the top of the Object Inspector shows the type of the object and allows you to choose a component. You might remove this list to save some space, considering that you can select components in the Object TreeView (by default placed on the top of the Object Inspector window).
- The properties that reference an object are now a different color and may be expanded without changing the selection.
- You can optionally view read-only properties in the Object Inspector. Of course, they are grayed out.
- The Object Inspector has a Properties dialog box, which allows you to customize the colors of the various types of properties and the overall behavior of this window.
- Since Delphi 5, the drop-down list for a property can include graphical elements. This feature is used for properties such as `Color` and `Cursor`, and is particularly useful for the `ImageIndex` property of components connected to an ImageList.

NOTE *Interface properties can now be configured at design time using the Object Inspector. This functionality uses the Interfaced Component Reference model introduced in Kylix/Delphi 6, where components can implement and hold references to interfaces as long as the interfaces are implemented by components. Interfaced Component References work like plain old component references, but interface properties can be bound to any component that implements the necessary interface. Unlike component properties, interface properties are not limited to a specific component type (a class or its derived classes). When you click the drop-down list in the Object Inspector editor for an interface property, all components on the current form (and linked forms) that implement the interface are shown.*

DROP-DOWN FONTS IN THE OBJECT INSPECTOR

The Delphi Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting, corresponding to the `Name` subproperty of the `Font` property. This capability is built into Delphi, but it has been disabled because most computers have a large number of fonts installed and rendering them can significantly slow the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplayFontNames` global variable of the VCLEditors unit. I've done this in the OiFontPk package, which you can find among the program examples for this chapter.

Once this package is installed, you can move to the `Font` property of any component and use the graphical Name drop-down menu, as displayed here:



Font (TFont)
Charset DEFAULT_CH
Color clWindowT
Height -11
Name Sans Serif

There is a second, more complex customization of the Object Inspector that I like and use frequently: a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations. See Appendix A to learn how to obtain this add-on package.

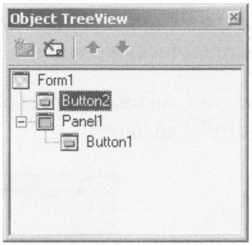
PROPERTY CATEGORIES

Delphi includes the idea of property categories, activated by the Arrange option of the Object Inspector's shortcut menu. If you set this option, properties are arranged by group rather than listed alphabetically, with each property possibly appearing in multiple groups. Categories have the benefit of reducing the complexity of the Object Inspector. You can use the View submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories). Although property categories have been available since Delphi 5, programmers rarely use them.

The Object TreeView

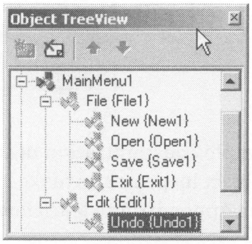
Delphi 5 introduced a TreeView for data modules, in which you could see the relations among nonvisual components, such as datasets, fields, actions, and so on. Delphi 6 extended the idea by providing an Object TreeView for every designer, including plain forms. The Object TreeView is placed by default above the Object Inspector.

The Object TreeView shows all the components and objects on the form in a tree representing their relations. The most obvious is the parent/child relation: If you place a panel on a form, a button inside the panel, and a button outside the panel, the tree will show one button under the form and the other under the panel:



Notice that the TreeView is synchronized with the Object Inspector and Form Designer. So, as you select an item and change the focus in any one of these three tools, the focus changes in the other two tools.

Besides parent/child, the Object TreeView shows other relations, such as owner/owned, component/subobject, and collection/item, plus various specific relations, including dataset/connection and data source/dataset relations. Here, you can see an example of the structure of a menu in the tree:

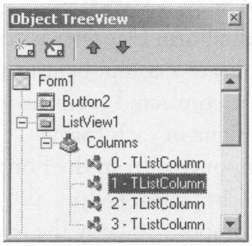


At times, the TreeView also displays "dummy" nodes, which do not correspond to an actual object but do correspond to a predefined object. As an example of this behavior, drop a Table component (from the BDE page); you'll see two grayed icons for the session and the alias. Technically, the Object TreeView uses gray icons for components that do not have design-time persistence. They are real components (at design time and at run time), but because they are default objects that are constructed at run time and have no persistent data that can be edited at design time, the Data Module Designer does not allow you to edit their properties. If you drop a Table on the form, you'll also see items that have next to them a red question mark enclosed in a yellow circle. This symbol indicates partially undefined items.

The Object TreeView supports multiple types of *dragging*:

- You can select a component from the palette (by clicking it, not dragging it), move the mouse over the tree, and click a component to drop it there. This technique allows you to drop a component in the proper container (form, panel, and others) regardless of the fact that its surface might be totally covered by other components—something that prevents you from dropping the component in the designer without first rearranging those components.
- You can drag components within the TreeView—for example, moving a component from one container to another. With the Form Designer, you can do so only with cut-and-paste techniques. Moving instead of cutting provides the advantage that any connections among components are not lost, as happens when you delete the component during the cut operation.
- You can drag components from the TreeView to the Diagram view, as you'll see later.

Right-clicking any element of the TreeView displays a shortcut menu similar to the component menu you get when the component is in a form (and in both cases, the shortcut menu may include items related to the custom component editors). You can even delete items from the tree. The TreeView also doubles as a collection editor, as you can see here for the `Columns` property of a ListView control. In this case, you can not only rearrange and delete items, but also add new items to the collection.



TIP *You can print the contents of the Object TreeView for documentation purposes. Simply select the window and use the File ➢ Print command (there is no Print command on the shortcut menu).*

Secrets of the Component Palette

The Component Palette is used to select components you want to add to the current designer. Move the mouse over a component and you'll see its name. In Delphi 7, the hint displays also the name of the unit that defines the component.

The Component Palette has many tabs—far too many, really. You may want to hide the tabs hosting components you don't plan to use and reorganize the Component Palette to suit your needs. In Delphi 7 you can also drag and drop the tabs to reorder them. Using the Palette page of the Environment Options dialog box, you can completely rearrange the components in the various pages, adding new elements or moving them from page to page.

When you have too many pages in the Component Palette, you'll need to scroll through them to reach a component. You can use a simple trick in this case: Rename the pages with shorter names, so all the pages will fit on the screen. (It's obvious—once you've thought about it.)

Delphi 7 offers another new feature. When there are too many components on a single page, Delphi displays a double down arrow; you click it to display the remaining components without having to scroll within the Palette page.

The Component Palette's shortcut menu has a Tabs submenu that lists all the palette pages in alphabetical order. You can use this submenu to change the active page, particularly when the page you need is not visible on the screen.

TIP *You can set the order of the entries in the Tabs submenu of the Component Palette shortcut menu to be the same as the order in the palette itself, rather than alphabetical. To do so, go to the* `Main Window` *Registry section of Delphi (under* `\Software\Borland\Delphi\7.0` *for the current user) and set the* `Sort Palette Tabs Menu` *key value to 0 (false).*

The significant undocumented feature of the Component Palette is "hot-track" activation. By setting special keys in the Registry, you can simply select a page of the palette by moving the mouse over the tab, without clicking. The same feature can be applied to the component scrollers on both sides of the palette, which appear when a page has too many components. To activate this hidden feature, add an `Extras` key under the `Borland\Delphi\7.0` key of the Registry in the `HKEY_CURRENT_USER\Software` section. Under this key, enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string '1'.

Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate a component in the form. During this operation, Delphi duplicates all the properties, keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

You can also copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source-code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
  Left = 152
  Top = 104
```

```
    Width = 75
    Height = 25
    Caption = 'My Button'
    TabOrder = 0
    Font.Name = 'Arial'
  end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

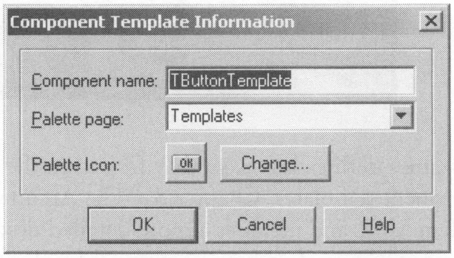
To use this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something that's completely incorrect and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This approach might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component ➢ Create Component Template menu command. This command opens the Component Template Information dialog box, where you enter the name of the template, the page of the Component Palette where it should appear, and an icon:



Component Template Information

Component name: TButtonTemplate

Palette page: Templates

Palette Icon:

By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

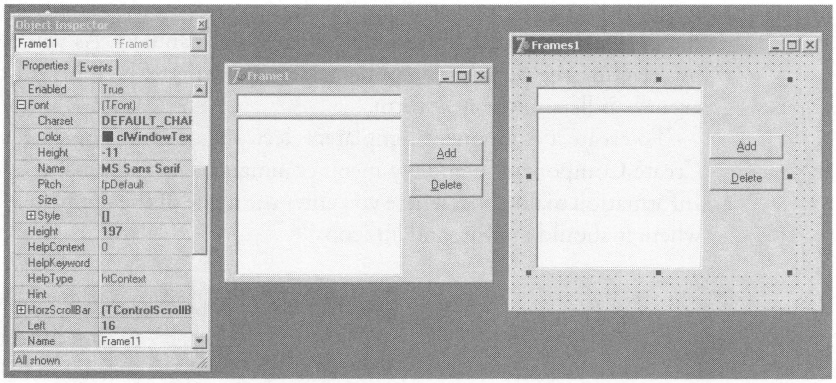
All the information about component templates is stored in a single file, `DELPHI32.DCT`, but there is no documented way to retrieve this information and edit a template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

TIP *A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry* `CCLibDir` *under the key* `\Software\Borland\Delphi\7.0\Component Templates`.

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the *frames* technology in Delphi does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready, you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each instance of the frame.

You can see a simple example, called Frames1, in Figure 1.8. A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames.



Frame11 TFrame1

Enabled	True
⊟Font	(TFont)
Charset	DEFAULT_CHAI
Color	■ clWindowTex
Height	-11
Name	MS Sans Serif
Pitch	fpDefault
Size	8
⊞Style	[]
Height	197
HelpContext	0
HelpKeyword	
HelpType	htContext
Hint	
⊞HorzScrollBar	(TControlScrollB
Left	16
Name	Frame11

FIGURE 1.8

The Frames1 example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.

Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than component templates. Chapter 8, "The Architecture of Delphi Applications," provides an in-depth look at VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful technique.

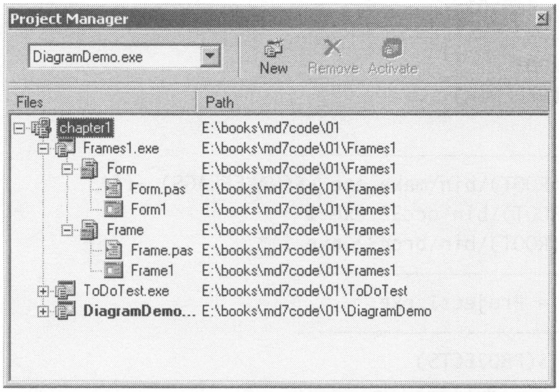
Managing Projects

Delphi's multitarget Project Manager (View ➢ Project Manager) works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files. All open packages will show up as projects in the Project Manager view, even if they haven't been added to the project group.

In Figure 1.9, you can see the Project Manager with a simple project group, including all the examples of the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on the projects in the group. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, and to open a unit.

FIGURE 1.9

Delphi's multitarget Project Manager



DiagramDemo.exe

Files	Path
chapter1	E:\books\md7code\01
Frames1.exe	E:\books\md7code\01\Frames1
Form	E:\books\md7code\01\Frames1
Form.pas	E:\books\md7code\01\Frames1
Form1	E:\books\md7code\01\Frames1
Frame	E:\books\md7code\01\Frames1
Frame.pas	E:\books\md7code\01\Frames1
Frame1	E:\books\md7code\01\Frames1
ToDoTest.exe	E:\books\md7code\01\ToDoTest
DiagramDemo...	E:\books\md7code\01\DiagramDemo

TIP *Since Delphi 6, the Project Manager also shows the open packages, even if they haven't been added to the project group. A* package *is a collection of components or other units compiled to a separate executable file, as you'll discover in Chapter 10 ("Libraries and Packages").*

Of all the projects in the group, only one is active; this is the project you operate on when you select a command such as Project ➢ Compile. The Project menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.) When you have multiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.

TIP *In Delphi 7, the Project Manager local menu allows you to compile projects beginning with a given one, using the Make All From Here or Build All From Here command. If you use this command on the first project of the group, you obtain the same effect as a Compile All or Build All command from the shortcut menu.*

Among the Project Manager's advanced features, you can drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project (drag-and-drop is also supported to open files in the code editor). You can easily see which project is selected and change it by using the combo box at the top of the Project Manager window, or by using the drop-down arrow next to the Run button on the Delphi toolbar.

Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu item, and choose Resource File (*.rc) as the file type. This resource file will be bound to the project automatically, even without a corresponding `$R` directive.

Delphi saves the project groups with the .BPG extension, which stands for Borland Project Group. This feature comes from C++Builder and from past Borland C++ compilers; this history is clearly visible when you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment. Here is a simple example:

```
#-----------------------------
VERSION = BWS.01
#-----------------------------
!ifndef ROOT
ROOT = $(MAKEDIR)\..
!endif
#-----------------------------
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)\bin\dcc32.exe $**
BRCC = $(ROOT)\bin\brcc32.exe $**
#-----------------------------
PROJECTS = Project1.exe
#-----------------------------
default: $(PROJECTS)
#-----------------------------
Project1.exe: Project1.dpr
  $(DCC)
```

Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. Instead, you can invoke the Project Options dialog from the Project Manager for each project. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of runtime packages.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{$`*X*`+}` and `{$`*X*`-}` directives, where you replace *X* with the option you want to set. This second approach is more flexible, because it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All project options are saved automatically with the project, but in a separate file with a .DOF extension. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a CFG file, for command-line compilation. The two files have similar content but a different format: The *dcc* command-line compiler cannot use .DOF files, but needs the .CFG format.

Another alternative for saving compiler options is to press Ctrl+O+O (press the O key twice while keeping Ctrl pressed). This key combination inserts, at the top of the current unit, compiler directives that correspond to the current project options (including all of the new compiler warning settings), as in the following listing:

```
{$A8,B-,C+,D+,E-,F-,G+,H+,I+,J-,K-,L+,M-,N+,O+,P+,Q-,R-,S-,T-,U-,V+,W-,X+,Y+,Z1}
{$MINSTACKSIZE $00004000}
{$MAXSTACKSIZE $00100000}
{$IMAGEBASE $00400000}
{$APPTYPE GUI}
{$WARN SYMBOL_DEPRECATED ON}
{$WARN SYMBOL_LIBRARY ON}
{$WARN SYMBOL_PLATFORM ON}
{$WARN UNIT_LIBRARY ON}
{$WARN UNIT_PLATFORM ON}
{$WARN UNIT_DEPRECATED ON}
{$WARN HRESULT_COMPAT ON}
{$WARN HIDING_MEMBER ON}
{$WARN HIDDEN_VIRTUAL ON}
{$WARN GARBAGE ON}
{$WARN BOUNDS_ERROR ON}
{$WARN ZERO_NIL_COMPAT ON}
{$WARN STRING_CONST_TRUNCED ON}
{$WARN FOR_LOOP_VAR_VARPAR ON}
{$WARN TYPED_CONST_VARPAR ON}
{$WARN ASG_TO_TYPED_CONST ON}
{$WARN CASE_LABEL_RANGE ON}
{$WARN FOR_VARIABLE ON}
{$WARN CONSTRUCTING_ABSTRACT ON}
{$WARN COMPARISON_FALSE ON}
{$WARN COMPARISON_TRUE ON}
{$WARN COMPARING_SIGNED_UNSIGNED ON}
{$WARN COMBINING_SIGNED_UNSIGNED ON}
{$WARN UNSUPPORTED_CONSTRUCT ON}
{$WARN FILE_OPEN ON}
{$WARN FILE_OPEN_UNITSRC ON}
{$WARN BAD_GLOBAL_SYMBOL ON}
{$WARN DUPLICATE_CTOR_DTOR ON}
{$WARN INVALID_DIRECTIVE ON}
{$WARN PACKAGE_NO_LINK ON}
{$WARN PACKAGED_THREADVAR ON}
{$WARN IMPLICIT_IMPORT ON}
{$WARN HPPEMIT_IGNORED ON}
{$WARN NO_RETVAL ON}
{$WARN USE_BEFORE_DEF ON}
{$WARN FOR_LOOP_VAR_UNDEF ON}
{$WARN UNIT_NAME_MISMATCH ON}
{$WARN NO_CFG_FILE_FOUND ON}
```

```
{$WARN MESSAGE_DIRECTIVE ON}
{$WARN IMPLICIT_VARIANTS ON}
{$WARN UNICODE_TO_LOCALE ON}
{$WARN LOCALE_TO_UNICODE ON}
{$WARN IMAGEBASE_MULTIPLE ON}
{$WARN SUSPICIOUS_TYPECAST ON}
{$WARN PRIVATE_PROPACCESSOR ON}
{$WARN UNSAFE_TYPE OFF}
{$WARN UNSAFE_CODE OFF}
{$WARN UNSAFE_CAST OFF}
```

Compiling and Building Projects

There are several ways to compile a project. If you run the project (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed. If you select Project ➢ Build All instead, every file is compiled, even if it has not changed. You should only need this second command infrequently, because Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options, in which case you have to use the Build All command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi Compiled Unit (DCU). (This step is performed only if the DCU file is not already up to date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option (in the Preferences page of the Environment Options dialog box).

WARNING *Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows a strange error message, the first thing you should try is the Build All command to resynchronize the make feature with the current files on disk.*

The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't run unless the proper DLLs (such as `vcl70.bpl`, which is quite large) are available on the computer where you want to run the program.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course, if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in Chapter 10.

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semicompiled" tools.

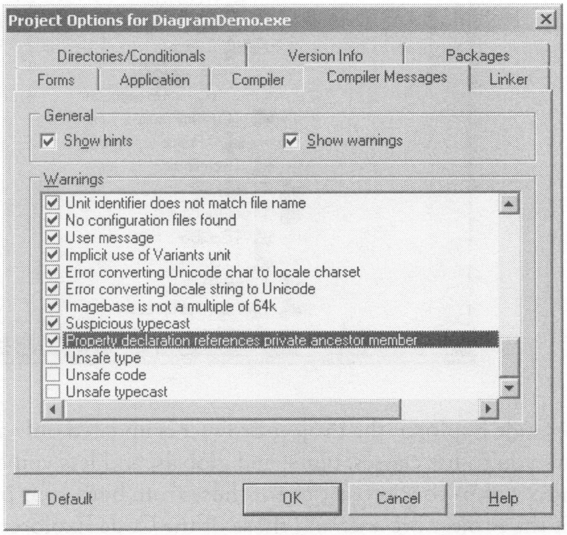
COMPILER MESSAGE HELPERS AND WARNINGS

As I mentioned at the beginning of this chapter (in the section "Extended Compiler Messages and Search Results in Delphi 7"), in addition to the classic compiler messages, Delphi 7 provides a new window with additional information about some error messages. This window is activated using the View ➢ Additional Message Info menu command. It displays information stored in a local file, which can be updated by downloading a new version from Borland's website.

Another change in Delphi 7 relates to the increased control you have over compiler warnings. The Project Options dialog box now includes a Compiler Messages page where you can choose many individual warnings. This feature was probably introduced due to the fact that Delphi 7 has a new set of warnings related to compatibility with the future Delphi for .NET tool. These warnings are quite extensive, and I've disabled them as shown in Figure 1.10.

FIGURE 1.10

The new Compiler Messages page of the Project Options dialog box



☑ Show hints
☑ Show warnings

☑ Unit identifier does not match file name
☑ No configuration files found
☑ User message
☑ Implicit use of Variants unit
☑ Error converting Unicode char to locale charset
☑ Error converting locale string to Unicode
☑ Imagebase is not a multiple of 64k
☑ Suspicious typecast
☑ Property declaration references private ancestor member
☐ Unsafe type
☐ Unsafe code
☐ Unsafe typecast

☐ Default

You can also enable or disable some of these warnings using compiler options like these:

```
{$Warn UNSAFE_CODE OFF}
{$Warn UNSAFE_CAST OFF}
{$Warn UNSAFE_TYPE OFF}
```

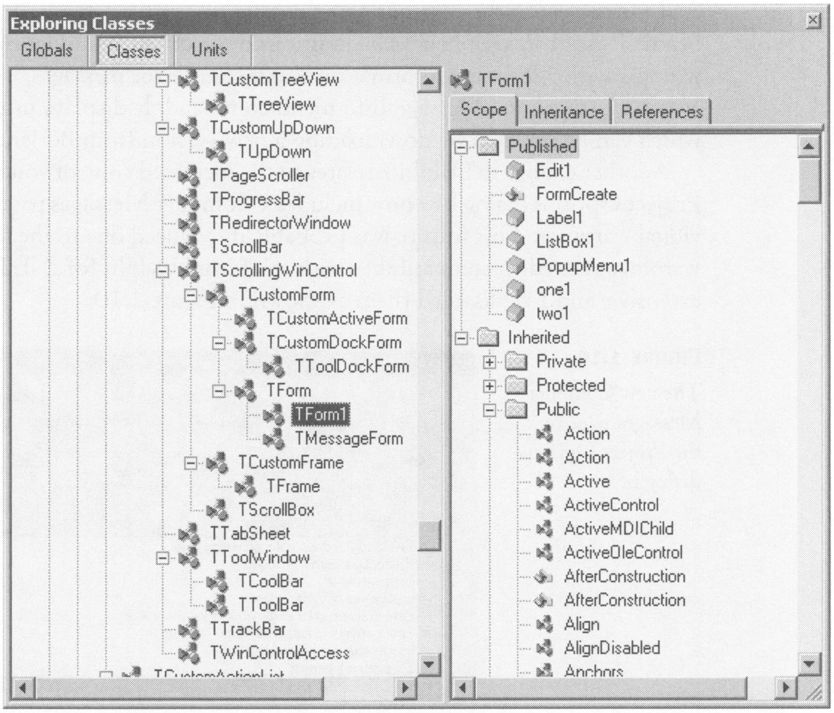
In general, it is better to keep these settings outside the source code of the program—something Delphi 7 finally allows you to do.

Exploring a Project's Classes

Delphi has always included a tool to browse the symbols of a compiled project, although this tool's name has changed many times (from Object Browser to Project Explorer and now to Project Browser). In Delphi 7, you activate the Project Browser window using the View ➢ Browser menu command, which displays the window shown in Figure 1.11. The browser allows you to see the hierarchical structure of the project's classes and to look for its symbols and the source-code lines where they are referenced.

FIGURE 1.11

The Project Browser


Exploring Classes
Globals
Classes
Units
TCustomTreeView
TTreeView
TCustomUpDown
TUpDown
TPageScroller
TProgressBar
TRedirectorWindow
TScrollBar
TScrollingWinControl
TCustomForm
TCustomActiveForm
TCustomDockForm
TToolDockForm
TForm
TForm1
TMessageForm
TCustomFrame
TFrame
TScrollBox
TTabSheet
TToolWindow
TCoolBar
TToolBar
TTrackBar
TWinControlAccess
TForm1
Scope
Inheritance
References
Published
Edit1
FormCreate
Label1
ListBox1
PopupMenu1
one1
two1
Inherited
Private
Protected
Public
Action
Action
Active
ActiveControl
ActiveMDIChild
ActiveOleControl
AfterConstruction
AfterConstruction
Align
AlignDisabled
Anchors

Unlike the Code Explorer, the Project Browser is updated only as you recompile the project. This browser allows you to list classes, units, and globals, and lets you choose whether to look only for symbols defined within your project or for those from both your project and VCL. You can change the settings of the Project Browser and those of the Code Explorer in the Explorer page of the Environment Options or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the categories you see in this window are specific to the Project Browser; others relate to both tools.

Additional and External Delphi Tools

In addition to the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (`PCE.exe`), and the Image Editor (`ImagEdit.exe`), are available from the Tools menu in the IDE. In addition, the Enterprise edition has a link to the SQL Monitor (`SqlMon.exe`). Other tools that are not directly accessible from the IDE include many command-line utilities you can find in the Delphi `bin` directory. For example, these tools include a command-line Delphi compiler (`DCC32.exe`), a Borland resource compiler (`BRC32.exe` and `BRCC32.exe`), and an executable viewer (`TDump.exe`).

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools, most of which are available in the `\Delphi7\bin` folder and in the Tools menu:

Web App Debugger (`WebAppDbg.exe`) The debugging web server introduced in Delphi 6. It is used to keep track of the requests sent to your applications and to debug them. This debugger was rewritten in Delphi 7: It is now a CLX application and its connectivity is based on sockets. I'll discuss this tool in Chapter 20.

XML Mapper (`XmlMapper.exe`) A tool for creating XML transformations to be applied to the format produced by the ClientDataSet component. You'll find more on this topic in Chapter 22.

External Translation Manager (`etm60.exe`) The stand-alone version of the Integrated Translation Manager. This external tool can be given to external translators and was available for the first time in Delphi 6.

Borland Registry Cleanup Utility (`D7RegClean.exe`) A tool that helps you remove all the Registry entries that Delphi 7 adds to a computer.

TeamSource An advanced version-control system provided with Delphi, starting with version 5. The tool is very similar to its past incarnation and is installed separately from Delphi. Delphi 7 ships with version 1.01 of Team Source, the same version available after applying an available patch to the Delphi 6 version.

WinSight (`Ws32.exe`) A Windows "message spy" program available in the `bin` directory.

Database Explorer A tool that can be activated from the Delphi IDE or as a stand-alone tool, using the `DBExplor.exe` program of the `bin` directory. Because it is meant for the BDE, the Database Explorer is not used much nowadays.

OpenHelp (`oh.exe`) The tool you can use to manage the structure of Delphi's own Help files, integrating third-party files into the help system.

Convert (`Convert.exe`) A command-line tool you can use to convert DFM files into the equivalent textual description and vice versa.

Turbo Grep (`Grep.exe`) A command-line search utility, which is much faster than the embedded Find In Files mechanism but not as easy to use.

Turbo Register Server (`TRegSvr.exe`) A tool you can use to register ActiveX libraries and COM servers. The source code for this tool is available under `\Demos\ActiveX\TRegSvr`.

Resource Explorer A powerful resource viewer (but not a full-blown resource editor) you can find under `\Demos\ResXplor`.

Resource Workshop An old 16-bit resource editor that can also manage Win32 resource files. The Delphi installation CD includes a separate installation for Resource Workshop. It was formerly included in Borland C++ and Pascal compilers for Windows and was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long filenames, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files.

The Files Produced by the System

Delphi produces various files for each project, and you should know what they are and how they are named. Basically, two elements have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you'll find in the directory where a Delphi project resides. The table also shows when or under what circumstances these files are created and their importance for future compilations.

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS

EXTENSION	FILE TYPE AND DESCRIPTION	CREATION TIME	REQUIRED TO COMPILE?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to endusers.
.CAB	The Microsoft Cabinet compressed-file format used for web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.

Continued on next page

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS *(continued)*

EXTENSION	**FILE TYPE AND DESCRIPTION**	**CREATION TIME**	**REQUIRED TO COMPILE?**
.DCP	Delphi Compiled Package: a file with symbol information for the code that was compiled into the package. It doesn't include compiled code, which is stored in DCU files or in the BPL file.	Compilation	Required when you use run-time packages. You'll distribute it only to other developers along with BPL files. You can compile an application with the units from a package just by having the DCP file and the BPL (and no DCU files).
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DDP	The Delphi Diagram Portfolio, used by the Diagram view of the editor (was .DTI in Delphi 5)	Development	No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
.~DF	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN	Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).	Development (ITE)	Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.
.DLL	Dynamic link library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK and now also .DPKW and .DPKL	Delphi Package: the project source code file of a package (or a specific project file for Windows or Linux).	Development	Yes.

Continued on next page

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS *(continued)*

EXTENSION	**FILE TYPE AND DESCRIPTION**	**CREATION TIME**	**REQUIRED TO COMPILE?**
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
.~DP	Backup of the Delphi Project file (`.DPR`).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.
.DSM	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.EXE	Executable file: the Windows application you've produced.	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or .HTML, for Hypertext Markup Language: the file format used for Internet web pages.	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
OCX	OLE Control Extension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
`.PAS`	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.

Continued on next page

TABLE 1.1: DELPHI PROJECT FILE EXTENSIONS *(continued)*

EXTENSION	FILE TYPE AND DESCRIPTION	CREATION TIME	REQUIRED TO COMPILE?
.~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.
.RPS	Translation Repository (part of the Integrated Translation Environment).	Development (ITE)	No. Required to manage the translations.
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
TODO	To-do list file, holding the items related to the entire project.	Development	No. This file hosts notes for the programmers.
.UDL	Microsoft Data Link.	Development	Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).

Besides the files generated during the development of a project in Delphi, many others are generated and used by the IDE itself. In Table 1.2, I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

TABLE 1.2: SELECTED DELPHI IDE CUSTOMIZATION FILE EXTENSIONS

EXTENSION	FILE TYPE
.DCI	Delphi code templates.
.DRO	Delphi's Object Repository. (The repository should be modified with the Tools ➢ Repository command.)
.DMT	Delphi menu templates.

Continued on next page

TABLE 1.2: SELECTED DELPHI IDE CUSTOMIZATION FILE EXTENSIONS ***(continued)***

EXTENSION	FILE TYPE
.DBI	Database Explorer information.
.DEM	Delphi edit mask (files with country-specific formats for edit masks).
.DCT	Delphi component templates.
.DST	Desktop settings file (one for each desktop setting you've defined).

Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time covering their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depends on syntax highlighting, but it isn't saved with the file. It is worth noting that there is a single file for the form's whole code, not just small code fragments.

TIP *In the listings in this book, I've matched the bold syntax highlighting of the editor for keywords and the italic for strings and comments.*

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a .DFM extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is by default a text representation of the form, but it can also be saved in a binary Windows Resource format. You can set the format you want to use for new projects in the Designer page of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command on a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This command closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can edit the textual description of a form, although you should do so with extreme care. As soon as you save the file, it will be parsed to regenerate the form. If you've made incorrect changes, compilation will stop with an error message; you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have good knowledge of Delphi programming.

TIP *In the book, I often show you excerpts of DFM files. In most of these excerpts, I show only the relevant components or properties; generally, I have removed the positional properties, the binary values, and other lines providing little useful information.*

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application: the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the Project ➢ View Source menu command.

Some of the other, less relevant files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF):

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

The Object Repository

Delphi has menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File ➢ New menu and in other pull-down menus. If you simply select File ➢ New ➢ Other, Delphi opens the Object Repository. You can use it to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New Items dialog box (shown in Figure 1.12) has several pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.

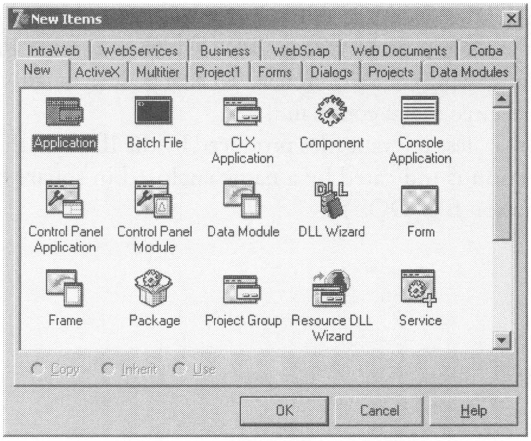

New Items
IntraWeb
WebServices
Business
WebSnap
Web Documents
Corba
New
ActiveX
Multitier
Project1
Forms
Dialogs
Projects
Data Modules
Application
Batch File
CLX Application
Component
Console Application
Control Panel Application
Control Panel Module
Data Module
DLL Wizard
Form
Frame
Package
Project Group
Resource DLL Wizard
Service
○ Copy ○ Inherit ○ Use
OK
Cancel
Help

FIGURE 1.12
The first page of the New Items dialog box, generally known as the Object Repository

TIP *The Object Repository has a shortcut menu you can use to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool—information that is particularly important when you're looking at wizards, projects, or forms that you've added to the Repository.*

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and "current project" pages). Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, and it will be ready to use later. Simply use the Project ➢ Add To Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Move to the form that you want to add, and select the Add To Repository command from its shortcut menu. Then, indicate the title, description, author, page, and icon in the resulting dialog box. Keep in mind that as you copy a project or form template to the Repository and then copy it back to another directory, you are simply doing a copy-and-paste operation; this isn't much different than copying the files manually.

THE EMPTY PROJECT TEMPLATE

When you start a new project, Delphi automatically opens a blank form, too. However, if you want to base a new project on one of the form objects or wizards, you don't need this form. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple:

1. Create a new project as usual.
2. Remove the project's only form.
3. Add this project to the templates, naming it *Empty Project*.

Continued on next page

THE EMPTY PROJECT TEMPLATE *(continued)*

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you have to remember to use the File ➢ Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools ➢ Repository command. This command opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing elements. You can even add new pages, rename or delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File ➢ New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File ➢ New Application), if no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File ➢ New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the `OBJREPOS` subdirectory of the Delphi main directory. At the same time, if you use a form or any other object directly without copying it, then you end up having some of your project files in this directory. It is important to realize how the Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files without copying data back and forth to the Repository.

INSTALLING NEW DLL WIZARDS

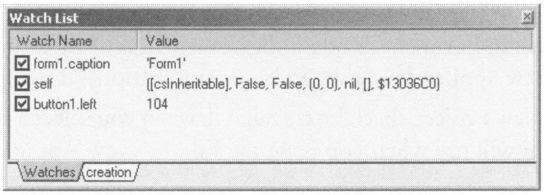
Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they are installed the same way you install a component or a package, using the Components ➢ Install Packages menu command and then clicking the Add button. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key `\Software\Borland\Delphi\7.0\Experts`. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.

Debugger Updates in Delphi 7

When you run a program in Delphi's IDE, you generally start it in the integrated debugger. You can set breakpoints, execute the code line by line, and explore its inner details, including the assembler code being executed and the use of CPU registries in the CPU view. I don't have space in this book to cover debugging in Delphi; see Appendix C for information about extra material on this topic. However, I do want to briefly mention a couple of new debugger features.

First, the Run Parameters dialog box in Delphi 7 allows you to set a working directory for the program being debugged. This means the current folder will be the one you indicate, not the one the program has been compiled into.

Another relevant change relates to the Watch List. It now has multiple tabs that let you keep different sets of active watches for different areas of the program you are debugging, without cluttering a single window. You can add a new group to the Watch List by using its shortcut menu, and also alter the visibility of column headers and enable individual watches with the corresponding check boxes.



Watch Name	Value
☑ form1.caption	'Form1'
☑ self	([csInheritable], False, False, (0, 0), nil, [], $13036C0)
☑ button1.left	104

NOTE *This book doesn't cover the Delphi debugger, but this is a very important topic. See the references to online material in Appendix C for information about how to download a free chapter that discusses debugging in Delphi.*

What's Next?

This chapter has presented an overview of the new and more advanced features of the Delphi 7 programming environment, including tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to begin *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

Now we are ready to spend the next chapter looking into the Delphi programming language. Then we'll proceed by studying the run-time library (RTL) and the class library included in Delphi.

Chapter 2

The Delphi Programming Language

THE DELPHI DEVELOPMENT ENVIRONMENT is based on an object-oriented extension of the Pascal programming language known as Object Pascal. Recently, Borland stated its intention to refer to the language as "the Delphi language," probably because the company wants to be able to say that Kylix uses the Delphi language and because Borland will provide the Delphi language on the Microsoft .NET platform. Due to years of habit, I'll use the two names interchangeably.

Most modern programming languages support *object-oriented programming* (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding). Although you can write Delphi code without understanding the core features of its language, you won't be able to master this environment until you fully understand the programming language.

NOTE *Due to space constraints and to the fact that the language hasn't changed much in recent years, in this chapter you'll find only a very fast-paced introduction to the language. You can read the more detailed description found in past editions of the book in the material available on my website (see Appendix C, "Free Companion Books on Delphi," for details). This material also includes* Essential Pascal*, a complete introduction to the standard Pascal language.*

The following topics are covered in this chapter:

- Classes and objects
- Encapsulation: *private* and *public*
- Using properties
- Constructors
- Objects and memory
- Inheritance
- Virtual methods and polymorphism
- Type-safe down-casting (run-time type information)

- Interfaces
- Working with exceptions
- Class references

Core Language Features

The Delphi language is an OOP extension of the classic Pascal language, which Borland pushed forward for many years with its Turbo Pascal compilers. The syntax of the Pascal language is known to be quite verbose and more readable than, for example, the C language. Its OOP extension follows the same approach, delivering the same power of the recent breed of OOP languages, from Java to C#.

Even the core language is subject to continuous changes, but few of them will affect your everyday programming needs. In Delphi 6, for example, Borland added support for several features more or less related to the development of Kylix, the Linux version of Delphi:

- A new directive for conditional compilation (`$IF`)
- A set of hint directives (`platform`, `deprecated`, and `library`, of which only the first is used to any extent) and the new `$WARN` directive used to disable them
- A `$MESSAGE` directive to emit custom information among compiler messages

Delphi 7 adds three additional compiler warnings: unsafe type, unsafe code, and unsafe cast. These warnings are emitted in case of operations that you won't be able to use to produce safe "managed" code on the Microsoft .NET platform (more on this in Chapter 25, "Delphi for .NET Preview: The Language and the RTL").

Another change relates to unit names, which can now be formed from multiple words separated by dot, as in the `marco.test` unit, saved in the `marco.test.pas` file. This feature will help support namespaces and more flexible unit references in Delphi for .NET and future versions of the Delphi compiler for Windows, but in Delphi 7 it has limited use.

Classes and Objects

Delphi is based on OOP concepts, and in particular on the definition of new class types. The use of OOP is partially enforced by the visual development environment, because for every new form defined at design time, Delphi automatically defines a new class. In addition, every component visually placed on a form is an object of a class type available in or added to the system library.

NOTE *The terms* class *and* object *are commonly used and often misused, so let's be sure we agree on their definitions. A* class *is a user-defined data type, which has a state (its representation or internal data) and some operations (its behavior or its methods). An* object *is an instance of a class, or a variable of the data type defined by the class. Objects are* actual *entities. When the program runs, objects take up some memory for their internal representation. The relationship between object and class is the same as the one between variable and type.*

As in most other modern OOP languages (including Java and C#), in Delphi a class-type variable doesn't provide the storage for the object, but is only a pointer or reference to the object in memory.

Before you use the object, you must allocate memory for it by creating a new instance or by assigning an existing instance to the variable:

```
var
  Obj1, Obj2: TMyClass;
begin
  // assign a newly created object
  Obj1 := TMyClass.Create;
  // assign to an existing object
  Obj2 := ExistingObject;
```

The call to `Create` invokes a default constructor available for every class, unless the class redefines it (as described later). To declare a new class data type in Delphi, with some local data fields and some methods, use the following syntax:

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

NOTE *The convention in Delphi is to use the letter* T *as a prefix for the name of every class you write and every other type (*T *stands for* Type*). This is just a convention—to the compiler,* T *is just a letter like any other—but it is so common that following it will make your code easier for other developers to understand.*

A method is defined with the *function* or *procedure* keyword, depending on whether it has a return value. Inside the class definition, methods can only be declared; they must be then defined in the *implementation* portion of the same unit. In this case, you prefix each method name with the name of the class it belongs to, using dot notation:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils.pas
  Result := IsLeapYear (Year);
end;
```

TIP *If you press Ctrl+Shift+C while the cursor is within the class definition, the Class Completion feature of the Delphi editor will generate the skeleton of the definition of the methods declared in a class.*

This is how you can use an object of the previously defined class:

```
var
  ADay: TDate;
begin
  // create an object
  ADay := TDate.Create;
  try
    // use the object
    ADay.SetValue (1, 1, 2000);
    if ADay.LeapYear then
      ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
  finally
    // destroy the object
    ADay.Free;
  end;
end;
```

Notice that `ADay.LeapYear` is an expression similar to `ADay.Year`, although the first is a function call and the second a direct data access. You can optionally add parentheses after the call of a function with no parameters. You can find the previous code snippets in the source code of the Dates1 example; the only difference is that the program creates a date based on the year provided in an edit box.

NOTE *The code snippet above uses a* `try/finally` *block to ensure the destruction of the object even in the case of exceptions in the code. You can find an introduction to the topic of exceptions at the end of this chapter.*

More on Methods

There is a lot more to say about methods. Here are some short notes about the features available in Delphi:

- Delphi supports method *overloading*. This means you can have two methods with the same name, provided that you mark the methods with the `overload` keyword and that the parameter lists of the two methods are sufficiently different. By checking the parameters, the compiler can determine which version you want to call.
- Methods can have one or more parameters with default values. If these parameters are omitted in the method call, they will be assigned the default value.
- Within a method, you can use the `Self` keyword to access the current object. When you refer to local data of the object, the reference to `self` is implicit. For example, in the `SetValue` method of the `TDate` class listed earlier, you use `Month` to refer to a field of the current object, and the compiler translates `Month` into `Self.Month`.
- You can define class methods, marked by the `class` keyword. A class method doesn't have an object instance to act upon, because it can be applied to an object of the class or to the class as a whole. Delphi doesn't (currently) have a way to define class data, but you can mimic this functionality by adding global data in the implementation portion of the unit defining the class.

- By default, methods use the `register` calling convention: (simple) parameters and return values are passed from the calling code to the function and back using CPU registers, instead of the stack. This process makes method calls much faster.

Creating Components Dynamically

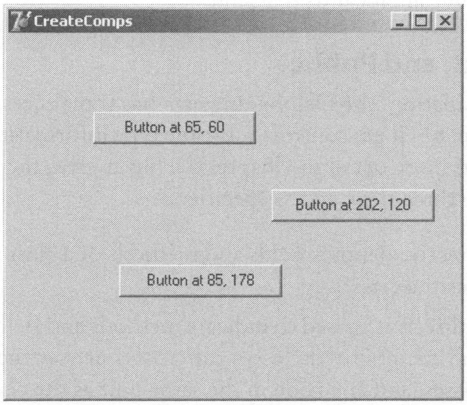
To emphasize the fact that Delphi components aren't much different from other objects (and also to demonstrate the use of the `Self` keyword), I've written the CreateComps example. This program has a form with no components and a handler for its `OnMouseDown` event, which I've chosen because it receives as a parameter the position of the mouse click (unlike the `OnClick` event). I need this information to create a button component in that position. Here is the method's code:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create (Self);
  Btn.Parent := Self;
  Btn.Left := X;
  Btn.Top := Y;
  Btn.Width := Btn.Width + 50;
  Btn.Caption := Format ('Button at %d, %d', [X, Y]);
end;
```

The effect of this code is to create buttons at mouse-click positions, as you can see in Figure 2.1. In the code, notice in particular the use of the `Self` keyword as both the parameter of the `Create` method (to specify the component's owner) and the value of the `Parent` property. I'll discuss these two elements (ownership and the `Parent` property) in Chapter 4, "Core Library Classes."

FIGURE 2.1

The output of the CreateComps example, which creates Button components at run time



When writing code like this, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't make any practical difference; but if there are multiple instances of a form, using `Form1` would be an error. In fact, if the `Form1` variable refers to the first form of that type being created, then by clicking in another form of the same type, the new button will always be displayed in the first form. The button's `Owner` and `Parent` will be `Form1`, not the form the user has clicked. In general, referring to a particular instance of a class when the current object is required is bad OOP practice.

Encapsulation

A class can have any amount of data and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated,* inside the class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, because the methods can check whether the date is valid and refuse to modify the new value if it is not. Encapsulation is important because it allows the class writer to modify the internal representation in a future version.

The concept of encapsulation is often indicated by the idea of a "black box." You don't know about the internals: You only know how to interface with the black box or use it regardless of its internal structure. The "how to use" portion, called the *class interface,* allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Of course, you are supposed to use methods to access the data, which is shielded from unauthorized access. This is the object-oriented approach to a classical programming concept known as *information hiding*. However, in Delphi there is the extra level of hiding, through properties, as we'll see later in this chapter.

Delphi implements this class-based encapsulation, but it still supports the classic module-based encapsulation using the structure of units. Every identifier that you declare in the interface portion of a unit becomes visible to other units of the program, provided there is a `uses` statement referring back to the unit that defines the identifier. On the other hand, identifiers declared in the implementation portion of the unit are local to that unit.

Private, Protected, and Public

For class-based encapsulation, the Delphi language has three access specifiers: `private`, `protected`, and `public`. A fourth, `published`, controls run-time type information (RTTI) and design-time information (as discussed in more detail in Chapter 4), but it gives the same programmatic accessibility as public. Here are the three *classic* access specifiers:

- The `private` directive denotes fields and methods of a class that are not accessible outside the unit that declares the class.
- The `protected` directive is used to indicate methods and fields with limited visibility. Only the current class and its inherited classes can access `protected` elements. More precisely, only the class, subclasses, and any code in the same unit as the class can access protected members, which opens up to a trick discussed in the sidebar "Accessing Protected Data of Other Classes (or, the "Protected Hack")" later in this chapter. We'll discuss this keyword again in the section "Protected Fields and Encapsulation."
- The `public` directive denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.

Generally, the fields of a class should be private and the methods public. However, this is not always the case. Methods can be private or protected if they are needed only internally to perform some partial computation or to implement properties. Fields might be declared as protected so that you can manipulate them in inherited classes, although this isn't considered a good OOP practice.

WARNING *Access specifiers only restrict code outside your unit from accessing certain members of classes declared in the interface section of your unit. This means that if two classes are in the same unit, there is no protection for their private fields.*

As an example, consider this new version of the `TDate` class:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

You might think of adding other functions, such as `GetDay`, `GetMonth`, and `GetYear`, which return the corresponding private data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class you are implementing.

Another new method is the `Increase` procedure, which increases the date by one day. This calculation is far from simple, because you need to consider the different lengths of the various months as well as leap and non–leap years. To make it easier to write the code, I'll change the internal implementation of the class to Delphi's `TDateTime` type for the internal implementation. The class definition will change to the following (the complete code is in the DateProp example):

```
type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Notice that because the only change is in the `private` portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

NOTE *The* `TDateTime` *type is a floating-point number. The integral portion of the number indicates the date since 12/30/1899, the same base date used by OLE Automation and Microsoft Win32 applications. (Use negative values to express previous years.) The decimal portion indicates the time as a fraction. For example, a value of 3.75 stands for the second of January 1900, at 6:00 A.M. (three-quarters of a day). To add or subtract dates, you can add or subtract the number of days, which is much simpler than adding days with a day/month/year representation.*

Encapsulating with Properties

Properties are a very sound OOP mechanism, or a well-thought-out application of the idea of encapsulation. Essentially, you have a name that completely hides its implementation details. This allows you to modify the class extensively without affecting the code using it. A good definition of properties is that of *virtual fields*. From the perspective of the user of the class that defines them, properties look exactly like fields, because you can generally read or write their value. For example, you can read the value of the `Caption` property of a button and assign it to the `Text` property of an edit box with the following code:

```
Edit1.Text := Button1.Caption;
```

It looks like you are reading and writing fields. However, properties can be directly mapped to data, as well as to access methods, for reading and writing the value. When properties are mapped to methods, the data they access can be part of the object or outside of it, and they can produce side effects, such as repainting a control after you change one of its values. Technically, a property is an identifier that is mapped to data or methods using a `read` and a `write` clause. For example, here is the definition of a `Month` property for a date class:

```
property Month: Integer read FMonth write SetMonth;
```

To access the value of the `Month` property, the program reads the value of the private field `FMonth`; to change the property value, it calls the method `SetMonth` (which must be defined inside the class, of course).

Different combinations are possible (for example, you could also use a method to read the value or directly change a field in the `write` directive), but the use of a method to change the value of a property is common. Here are two alternative definitions for the property, mapped to two access methods or mapped directly to data in both directions:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

Often, the actual data and access methods are private (or protected), whereas the property is public. For this reason, you must use the property to have access to those methods or data, a technique that provides both an extended and a simplified version of encapsulation. It is an *extended* encapsulation because not only can you change the representation of the data and its access functions, but you can also add or remove access functions without changing the calling code. A user only needs to recompile the program using the property.

TIP *When you're defining properties, take advantage of the extended class completion feature of Delphi's editor, which you activate with the Ctrl+Shift+C key combination. After you write the property name, type, and semicolon, press Ctrl+Shift+C, and Delphi will provide you with a complete definition and the skeleton of the* setter *method. Write* **`Get`** *in front of the name of the identifier after the* `read` *keyword, and you'll also have a* getter *method with almost no typing.*

PROPERTIES FOR THE *TDATE* CLASS

As an example, I've added properties for accessing the year, the month, and the day to an object of the `TDate` class discussed earlier. These properties are not mapped to specific fields, but they all map to the single `fDate` field storing the complete date information. This is why all the properties have both getter and setter methods:

```
type
  TDate = class
  public
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
```

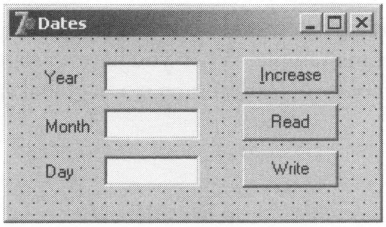
Each of these methods is easily implemented using functions available in the DateUtils unit (more details in Chapter 3, "The Run Time Library"). Here is the code for two of them (the others are very similar):

```
function TDate.GetYear: Integer;
begin
  Result := YearOf (fDate);
end;

procedure TDate.SetYear(const Value: Integer);
begin
  fDate := RecodeYear (fDate, Value);
end;
```

The code for this class is available in the DateProp example. The program uses a secondary unit for the definition of the `TDate` class to enforce encapsulation and creates a single-date object that is stored in a form variable and kept in memory for the entire execution of the program. Using a standard approach, the object is created in the form `OnCreate` event handler and destroyed in the form `OnDestroy` event handler. The program form (see Figure 2.2) has three edit boxes and buttons to copy the values of these edit boxes to and from the properties of the date object.

FIGURE 2.2

The DateProp example's form



WARNING *When writing the values, the program uses the* `SetValue` *method instead of setting each of the properties. Assigning the month and the day separately can cause you trouble when the month is not valid for the current day. For example, suppose the day is currently January 31, and you want to assign to it February 20. If you assign the month first, this part of the assignment will fail, because February 31 does not exist. If you assign the day first, the problem will arise when doing the reverse assignment. Due to the validity rules for dates, it is better to assign everything at once.*

ADVANCED FEATURES OF PROPERTIES

Properties have several advanced features I'll focus on in future chapters. Specifically, in Chapter 4 I'll cover the `TPersistent` class, RTTI, and streaming and I'll discuss writing custom Delphi components in Chapter 9, "Writing Delphi Components." Here is a short summary of these more advanced features:

- The `write` directive of a property can be omitted, making it a *read-only* property. The compiler will issue an error if you try to change the property value. You can also omit the `read` directive and define a *write-only* property, but that approach doesn't make much sense and is used infrequently.
- The Delphi IDE gives special treatment to *design-time* properties, which are declared with the `published` access specifier and generally displayed in the Object Inspector for the selected component. You'll find more on the `published` keyword and its effect in Chapter 4.
- An alternative is to declare properties, often called *run-time only* properties, with the `public` access specifier. These properties can be used in program code.
- You can define *array-based* properties, which use the typical notation with square brackets to access an element of a list. The *string list–based* properties, such as the `Lines` of a list box, are a typical example of this group.
- Properties have special directives, including `stored` and `default`, which control the *component streaming system* (introduced in Chapter 4 and detailed in Chapter 9).

NOTE *You can usually assign a value to a property or read it, and you can even use properties in expressions, but you cannot always pass a property as a parameter to a procedure or method. This is the case because a property is not a memory location, so it cannot be used as a* `var` *or* `out` *parameter; it cannot be passed by reference.*

Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a class's field, you only need to change the code of some methods of that class and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes.*

Let me clarify this idea with an example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of one of the forms:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

This approach works, but the data is connected to the entire program rather than a specific instance of the form. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```
type
  TForm1 = class(TForm)
```

```
public
  nClicks: Integer;
end;
```

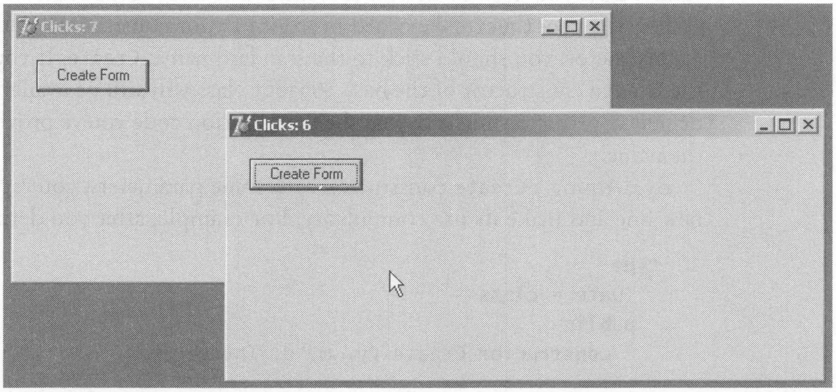
ADDING PROPERTIES TO FORMS

The previous class uses public data, so for the sake of encapsulation, you should instead change it to use private data and data-access functions. An even better solution is to add a property to the form. Every time you want to make some information of a form available to other forms, you should use a property, for all the reasons discussed in the section "Encapsulating with Properties." To do so, change the field declaration of the form (in the previous code) by adding the keyword `property` in front of it, and then press Ctrl+Shift+C to activate code completion. Delphi will automatically generate all the extra code you need.

The complete code for this form class is available in the FormProp example and illustrated in Figure 2.3. The program can create multi-instances of the form (that is, multiple objects based on the same form class), each with its own click count.

FIGURE 2.3

Two forms of the FormProp example at run time



NOTE *Notice that adding a property to a form doesn't add to the list of the form properties in the Object Inspector.*

In my opinion, properties should also be used in the form classes to encapsulate the access to the components of a form. For example, if you have a main form with a status bar used to display some information (and with the `SimplePanel` property set to True) and you want to modify the text from a secondary form, you might be tempted to write

```
Form1.StatusBar1.SimpleText := 'new text';
```

This is a standard practice in Delphi, but it's not a good one, because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (for example, replacing `StatusBar` with another control or activating multiple panels), you'll have to fix the code in many places. The alternative is to use a method or, even better, a property to hide the specific control. This property can be defined as

```
property StatusText: string read GetText write SetText;
```

with `GetText` and `SetText` methods that read from and write to the `SimpleText` property of the status bar (or the caption of one of its panels). In the program's other forms, you can refer to the form's `StatusText` property; and if the user interface changes, only the setter and getter methods of the property are affected.

NOTE *See Chapter 4 for a detailed discussion of how you can avoid having published form fields for components, which will improve encapsulation. But don't rush there: The description requires a good knowledge of Delphi, and the technique discussed has a few drawbacks.*

Constructors

So far, to allocate memory for objects, I've called the `Create` method. This is a *constructor*—a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later. All the data of the new instance is set to zero. If you want your instance data to start out with specific values, then you need to write a custom constructor to do that.

Use the `constructor` keyword in front of your constructor. Although you can use any name for a constructor, you should stick to the standard name, `Create`. If you use a name other than `Create`, the `Create` constructor of the base `TObject` class will still be available, but a programmer calling this default constructor might bypass the initialization code you've provided because they don't recognize the name.

By defining a `Create` constructor with some parameters, you replace the default definition with a new one and make its use compulsory. For example, after you define

```
type
  TDate = class
  public
    constructor Create (y, m, d: Integer);
```

you'll only be able to call this constructor and not the standard `Create`:

```
var
  ADay: TDate;
begin
  // Error, does not compile:
  ADay := TDate.Create;
  // OK:
  ADay := TDate.Create (1, 1, 2000);
```

The rules for writing constructors for custom components are different, as you'll see in Chapter 9. The reason is that in this case you have to override a virtual constructor. Overloading is particularly relevant for constructors, because you can add multiple constructors to a class and call them all `Create`; this approach makes the constructors easy to remember and follows a standard path provided by other OOP languages in which constructors must all have the same name. As an example, I've added to the class two separate `Create` constructors: one with no parameters, which hides the default constructor; and one with initialization values. The constructor with no parameter uses as the default value today's date (as you can see in the complete code of the DataView example):

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
```

Destructors and the *Free* Method

In the same way that a class can have a custom constructor, it can have a custom destructor—a method declared with the `destructor` keyword and called `Destroy`. Just as a constructor call allocates memory for the object, a destructor call frees the memory. Destructors are needed only for objects that acquire *external* resources in their constructors or during their lifetime. You can write custom code for a destructor, generally overriding the default `Destroy` destructor, to let an object execute some clean-up code before it is destroyed.

`Destroy` is a virtual destructor of the `TObject` class. You should never define a different destructor, because objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor of the specific class (virtual methods will be discussed later in this chapter).

`Free` is a method of the `TObject` class, inherited by all other classes. The `Free` method basically checks whether the current object (`Self`) is not `nil` before calling the `Destroy` virtual destructor. `Free` doesn't set the object to `nil` automatically; this is something you should do yourself! The object doesn't know which variables may be referring to it, so it has no way to set them all to `nil`.

Delphi 5 introduced a `FreeAndNil` procedure you can use to free an object and set its reference to `nil` at the same time. Call `FreeAndNil(Obj1)` instead of writing the following:

```
Obj1.Free;
Obj1 := nil;
```

NOTE *There's more on this topic in the section "Destroying Objects Only Once" later in this chapter.*

Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Delphi, instead, is based on an *object reference model*. The idea is that a variable of a class type, such as the `TheDay` variable in the preceding ViewDate example, does not hold the value of the object. Rather, it contains a reference, or a *pointer*, to indicate the memory location where the object has been stored. You can see this structure depicted in Figure 2.4.

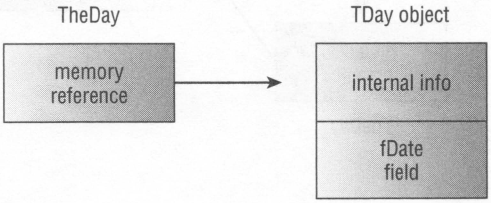


FIGURE 2.4
A representation of the structure of an object in memory, with a variable referring to it

The only problem with this approach is that when you declare a variable, you don't create an object in memory (which is inconsistent with all other variables, confusing new users of Delphi); you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of the components you place on a form are built automatically by the Delphi library.

You've seen how to create an instance of an object by applying a constructor to its class. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the `Free` method. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch. The object reference model has many consequences on assigning object and on managing memory, as you'll see in the next two sections.

Assigning Objects

If a variable holding an object only contains a reference to the object in memory, what happens if you copy the value of that variable? Suppose you write the `BtnTodayClick` method of the ViewDate example in the following way:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Create;
  TheDay := NewDay;
  LabelDate.Caption := TheDay.GetText;
end;
```

This code copies the memory address of the `NewDay` object to the `TheDay` variable (as shown in Figure 2.5); it doesn't copy the data of one object into the other. In this particular circumstance, this is not a very good approach—you keep allocating memory for a new object every time the button is clicked, but you never release the memory of the object the `TheDay` variable was previously pointing to.

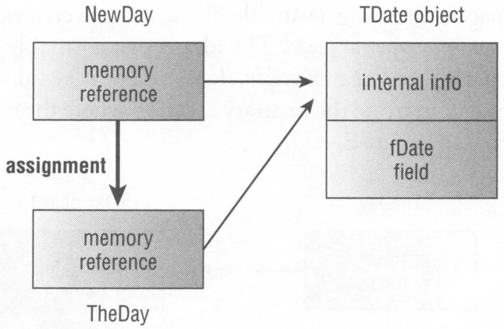


FIGURE 2.5 A representation of the operation of assigning an object reference to another object. This is different from copying the actual content of an object to another.

This specific issue can be solved by freeing the old object, as in the following code (which is also simplified, without the use of an explicit variable for the newly created object):

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  TheDay.Free;
  TheDay := TDate.Create;
```

The important thing to keep in mind is that, when you assign an object to another object, Delphi copies the reference to the object in memory to the new object reference. You should not consider this a negative: In many cases, being able to define a variable referring to an existing object can be a plus. For example, you can store the object returned by accessing a property and use it in subsequent statements, as this code snippet indicates:

```
var
  ADay: TDate;
begin
  ADay := UserInformation.GetBirthDate;
  // use a ADay
```

The same thing happens if you pass an object as a parameter to a function: You don't create a new object, but you refer to the same one in two different places in the code. For example, by writing this procedure and calling it as follows, you'll modify the `Caption` property of the `Button1` object, not of a copy of its data in memory (which would be totally useless):

```
procedure CaptionPlus (Button: TButton);
begin
  Button.Caption := Button.Caption + '+';
end;

// call...
CaptionPlus (Button1)
```

This means that the object is being passed by reference without the use of the `var` keyword and without any other obvious indication of the pass-by-reference semantic, which also confuses newcomers. What if you really want to change the data inside an existing object, so that it matches the data of another object? You have to copy each field of the object, which is possible only if they are all public, or provide a specific method to copy the internal data. Some classes of the VCL have an `Assign` method, which performs this copy operation. To be more precise, most of the VCL classes that inherit from `TPersistent`, but do not inherit from `TComponent`, have the `Assign` method. Other `TComponent`-derived classes have this method but raise an exception when it is called.

In the DateCopy example, I've added an `Assign` method to the `TDate` class and called it from the Today button, with the following code:

```
procedure TDate.Assign (Source: TDate);
begin
  fDate := Source.fDate;
end;
```

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
var
  NewDay: TDate;
begin
  NewDay := TDate.Create;
  TheDay.Assign(NewDay);
  LabelDate.Caption := TheDay.GetText;
  NewDay.Free;
end;
```

Objects and Memory

Memory management in Delphi is subject to three rules, at least if you allow the system to work in harmony without Access Violations and without consuming unneeded memory:

- Every object must be created before it can be used.
- Every object must be destroyed after it has been used.
- Every object must be destroyed only once.

Whether you must perform these operations in your code or can let Delphi handle memory management for you depends on the model you choose among the different approaches provided by Delphi.

Delphi supports three types of memory management for dynamic elements:

- Every time you create an object explicitly in your application code, you should also free it (with the only exception of a handful of system objects and of objects that are used through interface references). If you fail to do so, the memory used by that object won't be released for other objects until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form) becomes responsible for destroying all the objects it owns. In other words, when you free a form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it. This is the standard behavior of the components you create at design time by placing them on a form or data module. However, it is mandatory that you choose an owner that you can guarantee will be destroyed; for example, forms are generally owned by the global `Application` objects, which is destroyed by the library when the program ends.
- When Delphi's RTL allocates memory for strings and dynamic arrays, it will automatically free the memory when the reference goes out of scope. You don't need to free a string: When it becomes unreachable, its memory is released.

DESTROYING OBJECTS ONLY ONCE

If you call the `Free` method (or call the `Destroy` destructor) of an object twice, you get an error. However, if you remember to set the object to `nil`, you can call `Free` twice with no problem.

NOTE *You might wonder why you can safely call* `Free` *if the object reference is* `nil`*, but you can't call* `Destroy`*. The reason is that* `Free` *is a known method at a given memory location, whereas the virtual function* `Destroy` *is determined at run time by looking at the type of the object—a very dangerous operation if the object no longer exists.*

To sum things up, here are a couple of guidelines:

- Always call `Free` to destroy objects, instead of calling the `Destroy` destructor.
- Use `FreeAndNil`, or set object references to `nil` after calling `Free`, unless the reference is going out of scope immediately afterward.

In general, you can also check whether an object is `nil` by using the `Assigned` function. The following two statements are equivalent in most cases:

```
if Assigned (ADate) then ...
if ADate <> nil then ...
```

Notice that these statements test only whether the pointer is not `nil`; they do not check whether it is a valid pointer. If you write the following code, the test will be satisfied, and you'll get an error on the line with the call to the object method:

```
ToDestroy.Free;
if ToDestroy <> nil then
  ToDestroy.DoSomething;
```

It is important to realize that calling `Free` doesn't set the object to `nil`.

Inheriting from Existing Types

You'll often need to use a slightly different version of an existing class. For example, you might need to add a new method or slightly change an existing one. If you copy and paste the original class and then modify it (certainly a terrible programming practice, unless there is a specific reason to do so), you'll duplicate your code, bugs, and headaches. Instead, in such a circumstance you should use a key feature of OOP: *inheritance*.

To inherit from an existing class in Delphi, you only need to indicate that class at the beginning of the declaration of the new class. For example, this is done each time you create a new form:

```
type
  TForm1 = class(TForm)
  end;
```

This definition indicates that the `TForm1` class inherits all the methods, fields, properties, and events of the `TForm` class. You can call any public method of the `TForm` class for an object of the `TForm1` type. `TForm`, in turn, inherits some of its methods from another class, and so on, up to the `TObject` base class.

As an example of inheritance, you can derive a new class from `TDate` and modify its `GetText` function. You can find this code in the Dates unit of the NewDate example:

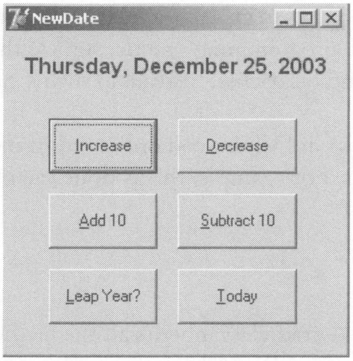
```
type
  TNewDate = class (TDate)
  public
    function GetText: string;
  end;
```

To implement the new version of the `GetText` function, I used the `FormatDateTime` function, which uses (among other features) the predefined month names available in Windows; these names depend on the user's regional and language settings. (Many of these regional settings are copied by Delphi into constants defined in the library, such as `LongMonthNames`, `ShortMonthNames`, and others you can find under the "Currency and date/time formatting variables" topic in the Delphi Help file.) Here is the `GetText` method, where *'dddddd'* stands for the long date format:

```
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

TIP *Using regional information, the NewDate program automatically adapts itself to different Windows user settings. If you run this program on a computer with regional settings referring to a language other than English, it will automatically show month names in that language. To test this behavior, you just need to change the regional settings. Notice that regional-setting changes immediately affect the running programs.*

Once you have defined the new class, you need to use this new data type in the code of the form of the NewDate example, defining the `TheDay` object of type `TNewDate` and creating an object of this new class in the `FormCreate` method. You don't have to change the code with method calls, because the inherited methods still work exactly in the same way; however, their effect changes, as the new output demonstrates (see Figure 2.6).



Thursday, December 25, 2003

FIGURE 2.6
The output of the NewDate program, with the name of the month and of the day depending on Windows regional settings

Protected Fields and Encapsulation

The code of the `GetText` method of the `TNewDate` class compiles only if it is written in the same unit as the `TDate` class. In fact, it accesses the `fDate` private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the `fDate` field as protected or add a protected access method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write inherited classes.

However, this approach violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If 10 derived classes are accessing this data, changing its definition means potentially modifying the code in each of the 10 classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate solution can be obtained by using a virtual method, a topic I'll discuss in detail in the section "Late Binding and Polymorphism." If you choose not to use encapsulation in order to obtain faster coding of the inherited classes, then your design might not follow the object-oriented principles.

ACCESSING PROTECTED DATA OF OTHER CLASSES (OR, THE "PROTECTED HACK")

You've seen that in Delphi, the private and protected data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this class (part of the Protection example):

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  end;
```

Once you place this class in its own unit, you won't be able to access its protected portion from other units directly. Accordingly, if you write the following code

```
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.ProtectedData := 20;  // won't compile
```

the compiler will issue an error message, "Undeclared identifier: 'ProtectedData.'" At this point, you might think there is no way to access the protected data of a class defined in a different unit. However, there is a way. Consider what happens if you create an apparently useless derived class, such as the following:

```
type
  TTestHack = class (TTest);
```

Now, if you make a direct cast of the object to the new class and access the protected data through it, this is how the code will look:

```
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  TTestHack (Obj).ProtectedData := 20; // compiles!
```

Continued on next page

ACCESSING PROTECTED DATA OF OTHER CLASSES (OR, THE "PROTECTED HACK") ***(continued)***

This code compiles and works properly, as you can see by running the Protection program. How is it possible for this approach to work? Well, if you think about it, the `TTestHack` class automatically inherits the protected fields of the `TTest` base class, and because the `TTestHack` class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible. As you would expect, if you move the declaration of the `TTestHack` class to a secondary unit, the program will no longer compile.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are times when using this technique is the best solution, as you'll see by looking at the VCL source code and the code of many Delphi components. Two examples that come to mind are accessing the `Text` property of the `TControl` class and the `Row` and `Col` positions of the `DBGrid` control. These two ideas are demonstrated by the TextProp and DBGridCol examples, respectively. (These examples are quite advanced, so I suggest that only programmers with a good background in Delphi programming read them at this point in the text—other readers might come back later.) Although the first example shows a reasonable example of using the typecast *cracker*, the DBGrid example of `Row` and `Col` is a counterexample—it illustrates the risks of accessing bits that the class writer chose not to expose. The row and column of a `DBGrid` do not mean the same thing as they do in a `DrawGrid` or `StringGrid` (the base classes). First, `DBGrid` does not count the fixed cells as actual cells (it distinguishes data cells from decoration), so your row and column indexes will have to be adjusted by whatever decorations are currently in effect on the grid (and those can change on the fly). Second, the `DBGrid` is a virtual view of the data. When you scroll up in a `DBGrid`, the data may move underneath it, but the currently selected row might not change.

This technique—declaring a local type only so that you can access protected data members of a class—is often described as a *hack*, and it should be avoided whenever possible. The problem is not accessing protected data of a class in the same unit but declaring a class for the sole purpose of accessing protected data of an existing object of a different class! The danger of this technique is in the hard-coded typecast of an object from a class to a different one.

Inheritance and Type Compatibility

Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a Boolean variable, unless you use an explicit typecast. The rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type refers to a single type definition. To simplify your life, Delphi makes some predefined types assignment compatible: you can assign an `Extended` to a `Double` and vice versa, with automatic promotion or demotion (and potential accuracy loss).

WARNING *If you redefine the same data type in two different units, the types won't be compatible, even if their names are identical. A program using two equally named types from two different units will be a nightmare to compile and debug.*

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. You can do so because a dog is an animal! As a general rule, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
var
  MyAnimal: TAnimal;
  MyDog: TDog;
begin
  MyAnimal := MyDog;  // This is OK
  MyDog := MyAnimal;  // This is an error!!!
```

Late Binding and Polymorphism

Pascal functions and procedures are usually based on *static* or *early binding*. This means that a method call is resolved by the compiler and linker, which replace the request with a call to the specific memory location where the function or procedure resides (the routine's address). OOP languages allow the use of another form of binding, known as *dynamic* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

This technique is known as *polymorphism* (a Greek word meaning "many forms"). Polymorphism means you can call a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the class of the object the variable refers to, because of the type-compatibility rule discussed in the previous section. The advantage of polymorphism is being able to write simpler code, treating disparate object types as if they were the same and getting the correct runtime behavior.

For example, suppose that a class and an inherited class (let's say `TAnimal` and `TDog`) both define a method, and this method has late binding. You can apply this method to a generic variable, such as `MyAnimal`, which at run time can refer either to an object of class `TAnimal` or to an object of class `TDog`. The actual method to call is determined at run time, depending on the class of the current object.

The PolyAnimals example demonstrates this technique. The `TAnimal` and `TDog` classes have a `Voice` method that outputs the sound made by the selected animal, both as text and as sound (using a call to the `PlaySound` API function defined in the MMSystem unit). The `Voice` method is defined as virtual in the `TAnimal` class and is later overridden when you define the `TDog` class, by the use of the `virtual` and `override` keywords:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

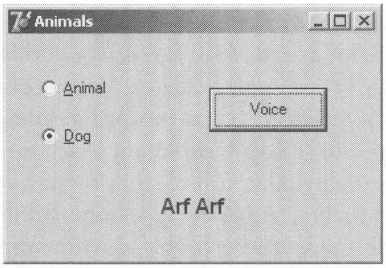
The effect of the call `MyAnimal.Voice` depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice`, instead. This happens only because the function is virtual (you can experiment by removing this keyword and recompiling).

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined in other units—or that haven't been written yet! The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` inherited classes.

NOTE *This is the key technical reason why object-oriented programming languages favor reusability. You can write code that uses classes within a hierarchy without any knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition: The ancestor classes of the hierarchy need to be designed very carefully.*

In Figure 2.7, you can see an example of the output of the PolyAnimals program. By running it, you'll also hear the corresponding sounds produced by the `PlaySound` API call.

FIGURE 2.7
The output of the PolyAnimals example



Arf Arf

Overriding and Redefining Methods

As you have just seen, to override a late-bound method in a descendant class, you need to use the `override` keyword. Note that this can take place only if the method was defined as virtual (or dynamic) in the ancestor class. Otherwise, if it is a static method, there is no way to activate late binding, other than to change the code of the ancestor class.

The rules are simple: A method defined as static remains static in every inherited class, unless you hide it with a new virtual method having the same name. A method defined as virtual remains late-bound in every inherited class (unless you hide it with a static method, which is quite a foolish thing to do). There is no way to change this behavior, because of the way the compiler generates different code for late-bound methods.

To redefine a static method, you add a method to an inherited class having the same parameters or different parameters than the original one, without any further specifications. To override a virtual method, you must specify the same parameters and use the `override` keyword:

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;
```

```
TMyDerivedClass = class (MyClass)
  procedure One; override;
  procedure Two;
end;
```

Typically, you can override a method two ways: replace the method of the ancestor class with a new version, or add more code to the existing method. This can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write:

```
procedure TMyDerivedClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

When you override an existing virtual method of a base class, you must use the same parameters. When you introduce a new version of a method in a descendent class, you can declare it with the parameters you want. In fact, this will be a new method unrelated to the ancestor method of the same name—they only happen to use the same name. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMyDerivedClass = class (TMyClass)
    procedure One (S: string);
  end;
```

NOTE Using the class definitions just given, when you create an object of the TMyDerivedClass class, you can call its `One` *method with the string parameter, but not the parameter-less version defined in the base class. If this is what you need, it can be accomplished by marking the redeclared method (the one in the derived class) with the* `overload` *keyword. If the method has different parameters than the version in the base class, it becomes effectively an overloaded method; otherwise it replaces the base class method. Notice that the method doesn't need to be marked with* `overload` *in the base class. However, if the method in the base class is virtual, the compiler issues the warning "Method 'One' hides virtual method of base type 'TMyClass.'" To avoid this message and to instruct the compiler more precisely on your intentions, you can use the* `reintroduce` *directive. If you are interested in this advanced topic, you can find this code in the Reintr example and experiment with it further.*

Virtual versus Dynamic Methods

In Delphi, there are two ways to activate late binding. You can declare the method as virtual, as you have seen, or declare it as dynamic. The syntax of the `virtual` and `dynamic` keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

Virtual methods are based on a *virtual method table* (VMT, also known as a *vtable*), which is an array of method addresses. For a call to a virtual method, the compiler generates code to jump to an address stored in the *n*th slot in the object's virtual method table. VMTs allow fast execution of the method calls, but they require an entry for each virtual method for each descendant class, even if the method is not overridden in the inherited class.

`Dynamic` method calls, on the other hand, are dispatched using a unique number indicating the method, which is stored in a class only if the class defines or overrides it. The search for the corresponding function is generally slower than the one-step table lookup for virtual methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method.

MESSAGE HANDLERS

A late-bound method can be used to handle a Windows message, too, although the technique is somewhat different. For this purpose Delphi provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter. The `message` directive is followed by the number of the Windows message the method wants to handle.

WARNING *The `message` directive is also available in Kylix and is fully supported by the language and the run-time library (RTL). However, the visual portion of the CLX application framework does not use message methods to dispatch notifications to controls. For this reason, whenever possible, you should use a virtual method provided by the library rather than handle a Windows message directly. Of course, this matters only if you want your code to be more portable.*

For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `wm_User` Windows constant:

```
type
  TForm1 = class(TForm)
    ...
    procedure WMUser (var Msg: TMessage);
      message wm_User;
  end;
```

The name of the procedure and the type of the parameters are up to you, although there are several predefined record types for the various Windows messages. You could later generate this message, invoking the corresponding method, by writing:

```
PostMessage (Form1.Handle, wm_User, 0, 0);
```

This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions. You can also dispatch a message immediately by calling the SendMessage API or the VCL Perform method.

Abstract Methods

The `abstract` keyword is used to declare methods that will be defined only in inherited classes of the current class. The `abstract` directive fully defines the method; it is not a forward declaration. If you try to provide a definition for the method, the compiler will complain. In Delphi, you can create instances of classes that have abstract methods. However, when you try to do so, Delphi's 32-bit compiler issues the warning message "Constructing instance of <class name> containing abstract methods." If you happen to call an abstract method at run time, Delphi will raise an exception, as demonstrated by the AbstractAnimals example (an extension of the PolyAnimals example), which uses the following class:

```
type
  TAnimal = class
  public
    function Voice: string; virtual; abstract;
```

NOTE *Most other OOP languages use a stricter approach: you cannot generally create instances of classes containing abstract methods.*

You might wonder why you would want to use abstract methods. The reason lies in the use of polymorphism. If class `TAnimal` has the virtual method `Voice`, every inherited class *can* redefine it. If it has the abstract method `Voice`, every inherited class *must* redefine it.

In early versions of Delphi, if a method overriding an abstract method called `inherited`, the result was in an abstract method call. Since Delphi 6, the compiler has been enhanced to notice the presence of the abstract method and skip the `inherited` call. This means you can safely use `inherited` in every overridden method, unless you specifically want to disable executing some code of the base class.

Type-Safe Down-Casting

The Delphi type-compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned earlier, the reverse is not possible. Now, suppose the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, it should be possible to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, you could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods you are calling actually exist.

To solve the problem, you can use techniques based on *run-time type information* (RTTI, for short). Essentially, because each object "knows" its type and its parent class, you can ask for this information with the `is` operator (or in some peculiar cases using the `InheritsFrom` method of `TObject`). The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then ...
```

The `is` expression evaluates as True only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from `TDog`. This means that if you test whether a `TDog` object is of type `TAnimal`, the test will succeed. In other words, this expression evaluates as True if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

Now that you know for sure that the animal is a dog, you can make a safe typecast (or type conversion). You can accomplish this direct cast by writing the following code:

```
var
  MyDog: TDog;
```

```
begin
  if MyAnimal is TDog then
  begin
    MyDog := TDog (MyAnimal);
    Text := MyDog.Eat;
  end;
```

This same operation can be accomplished directly by the second RTTI operator, `as`, which converts the object only if the requested class is compatible with the actual one. The parameters of the `as` operator are an object and a class type, and the result is an object converted to the new class type. You can write the following snippet:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

If you only want to call the `Eat` function, you might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second approach raises an exception if the object type is incompatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described at the end of this chapter).

To avoid this exception, use the `is` operator and, if it succeeds, make a plain typecast (in fact, there is no reason to use `is` and `as` in sequence, doing the type check twice):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both RTTI operators are very useful in Delphi because you often want to write generic code that can be used with several components of the same type or even of different types. When a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`); so, you often need to cast it back to the original component type:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

This is a common technique in Delphi, and I'll use it in examples throughout the book. The two RTTI operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism first. Only in special cases, where polymorphism alone cannot be applied, should you try using the RTTI operators to complement it. *Do not use RTTI instead of polymorphism.* This is bad programming practice, and it results in slower programs. RTTI has a negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As you have seen, virtual method calls require just a memory lookup, which is much faster.

NOTE *Run-time type information (RTTI) involves more than the* `is` *and* `as` *operators. You can access detailed class and type information at run time, particularly for published properties, events, and methods. You'll find more on this topic in Chapter 4.*

Using Interfaces

When you define an abstract class to represent the base class of a hierarchy, you can come to a point at which the abstract class is so abstract that it only lists a series of virtual functions without providing any implementation. This kind of *purely abstract class* can also be defined using a specific technique: an interface. For this reason, we refer to these classes as *interfaces.*

Technically, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element with distinctive features:

- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to the way Delphi manages long strings and makes memory management almost automatic.
- A class can inherit from a single base class, but it can implement multiple interfaces.
- Just as all classes descend from `TObject`, all interfaces descend from `IInterface`, forming a totally separate hierarchy.

NOTE *The base interface class was* `IUnknown` *until Delphi 5, but Delphi 6 introduced a new name for it—*`IInterface`*—to mark even more clearly the fact that this language feature is separate from Microsoft's COM (which uses* `IUnknown` *as its base interface). Delphi interfaces are available also in Kylix.*

It is important to note that interfaces support a slightly different OOP model than classes. Interfaces provide a less restricted implementation of polymorphism. Object reference polymorphism is based around a specific branch of a hierarchy. Interface polymorphism works across an entire hierarchy. Certainly, interfaces favor encapsulation and provide a looser connection between classes than inheritance. Notice that the most recent OOP languages, from Java to C#, have the notion of interfaces.

Here is the syntax of the declaration of an interface (which, by convention, starts with the letter *I*):

```
type
  ICanFly = interface
    ['{EAD9C4B4-E1C5-4CF4-9FA0-3B812C880A21}']
    function Fly: string;
  end;
```

This interface has a GUID (Globally Unique Identifier)—a numeric ID following its declaration and based on Windows conventions. You can generate these identifiers by pressing Ctrl+Shift+G in the Delphi editor.

NOTE *Although you can compile and use an interface without specifying a GUID for it, you'll generally want to generate the GUID because it is required to perform interface querying or dynamic* `as` *typecasts using that interface type. The whole point of interfaces is (usually) to take advantage of greatly extended type flexibility at run time; so, compared with class types, interfaces without GUIDs are not very useful.*

Once you've declared an interface, you can define a class to implement it:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

The RTL already provides a few base classes to implement the basic behavior required by the `IInterface` interface. For internal objects, use the `TInterfacedObject` class I've used in this code.

You can implement interface methods with static methods (as in the previous code) or with virtual methods. You can override virtual methods in inherited classes by using the `override` directive. If you don't use virtual methods, you can still provide a new implementation in an inherited class by redeclaring the interface type in the inherited class, rebinding the interface methods to new versions of the static methods. At first sight, using virtual methods to implement interfaces seems to allow for smoother coding in inherited classes, but both approaches are equally powerful and flexible. However, the use of virtual methods affects code size and memory.

NOTE *The compiler has to generate stub routines to fix up the interface call entry points to the matching method of the implementing class, and adjust the* `self` *pointer. The interface method stubs for static methods have to adjust* `self` *and jump to the real method in the class. The interface method stubs for virtual methods are much more complicated, requiring about four times more code (20 to 30 bytes) in each stub than the static case. Also, adding more virtual methods to the implementing class just bloats the virtual method table (VMT) that much more in the implementing class and all its descendents. An interface already has its own VMT, and redeclaring an interface in descendents to rebind the interface to new methods in the descendent is just as polymorphic as using virtual methods, but much smaller in code size.*

Now that you have defined an implementation of the interface, you can write some code to use an object of this class, through an interface-type variable:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

As soon as you assign an object to an interface-type variable, Delphi automatically checks to see whether the object implements that interface, using the `as` operator. You can explicitly express this operation as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

NOTE *The compiler generates different code for the* `as` *operator when used with interfaces or with classes. With classes, the compiler introduces run-time checks to verify that the object is effectively "type-compatible" with the given class. With interfaces, the compiler sees at compile time that it can extract the necessary interface from the available class type, so it does. This operation is like a "compile-time* `as`*," not something that exists at run time.*

Whether you use the direct assignment or the `as` statement, Delphi does one extra thing: It calls the `_AddRef` method of the object (defined by `IInterface`). The standard implementation of this method, like the one provided by `TInterfacedObject`, is to increase the object's reference count. At the same time, as soon as the `Flyer1` variable goes out of scope, Delphi calls the `_Release` method (again part of `IInterface`). The `TInterfacedObject`'s implementation of `_Release` decreases the reference count, checks whether the reference count is zero, and, if necessary, destroys the object. For this reason, the previous example doesn't include any code to free the object you've created.

In other words, in Delphi, objects referenced by interface variables are reference-counted, and they are automatically de-allocated when no interface variable refers to them any more.

WARNING *When using interface-based objects, you should generally access them only with object references or only with interface references. Mixing the two approaches breaks the reference counting scheme provided by Delphi and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables. If you want to be able to mix them, instead, disable the reference counting by writing your own base class instead of using* `TInterfacedObject`.

Working with Exceptions

Another key feature of Delphi is its support for *exceptions*. Exceptions make programs more robust by providing a standard way for notifying and handling errors and unexpected conditions. Exceptions make programs easier to write, read, and debug because they allow you to separate the error-handling code from your normal code, instead of intertwining the two. Enforcing a logical split between code and error handling and branching to the error handler automatically makes the actual logic cleaner and clearer. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

At run time, Delphi libraries raise exceptions when something goes wrong (in the run-time code, in a component, or in the operating system). From the point in the code at which it is raised, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, the VCL handles it, by displaying a standard error message and then trying to continue the program by handling the next system message or user request.

The whole mechanism is based on four keywords:

try Delimits the beginning of a protected block of code.

except Delimits the end of a protected block of code and introduces the exception-handling statements.

finally Specifies blocks of code that must always be executed, even when exceptions occur. This block is generally used to perform cleanup operations that should always be executed, such as closing files or database tables, freeing objects, and releasing memory and other resources acquired in the same program block.

raise Generates an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also raise exceptions in your own code when it discovers invalid or inconsistent data at run time. The `raise` keyword can also be used inside a handler to *re-raise* an exception; that is, to propagate it to the next handler.

TIP *Exception handling is no substitute for proper control flow within a program. Keep using* `if` *statements to test user input and other foreseeable error conditions. You should use exceptions only for abnormal or unexpected situations.*

Program Flow and the *finally* Block

The power of exceptions in Delphi relates to the fact that they are "passed" from a routine or method to the caller, up to a global handler (if the program provides one, as Delphi applications generally do), instead of following the standard execution path of the program. So the real problem you might have is not how to stop an exception but how to execute code even if an exception is raised.

Consider this code, which performs some time-consuming operations and uses the hourglass cursor to show the user that it's doing something:

```
Screen.Cursor := crHourglass;
// long algorithm...
Screen.Cursor := crDefault;
```

In case there is an error in the algorithm (as I've included on purpose in the TryFinally example's event handlers), the program will break, but it won't reset the default cursor. This is what a `try/finally` block is for:

```
Screen.Cursor := crHourglass;
try
  // long algorithm...
finally
  Screen.Cursor := crDefault;
end;
```

When the program executes this function, it always resets the cursor, regardless of whether an exception (of any sort) occurs.

This code doesn't handle the exception; it merely makes the program robust in case an exception is raised. A `try` block can be followed by either an `except` or a `finally` statement, but not both of them at the same time; so, if you want to also handle the exception, the typical solution is to use two nested `try` blocks. You associate the internal block with a `finally` statement and the external block with an `except` statement, or vice versa as the situation requires. Here is the skeleton of the code for the third button in the TryFinally example:

```
Screen.Cursor := crHourglass;
try try
  // long algorithm...
finally
  Screen.Cursor := crDefault;
end;
except
  on E: EDivByZero do ...
end;
```

Every time you have some finalization code at the end of a method, you should place the code in a `finally` block. You should always, invariably, and continuously (how can I stress this more?) protect your code with `finally` statements, to avoid resource or memory leaks in case an exception is raised.

TIP Handling the exception is generally much less important than using `finally` *blocks, because Delphi can survive most exceptions. Too many exception-handling blocks in your code probably indicate errors in the program flow and possibly a misunderstanding of the role of exceptions in the language. In the examples in the rest of the book, you'll see many* `try/finally` *blocks, a few* `raise` *statements, and almost no* `try/except` *blocks.*

Exception Classes

In the exception-handling statements shown earlier, you caught the `EDivByZero` exception, which is defined by Delphi's RTL. Other such exceptions refer to run-time problems (such as a wrong dynamic cast), Windows resource problems (such as out-of-memory errors), or component errors (such as a wrong index). Programmers can also define their own exceptions; you can create a new inherited class of the default exception class or one of its inherited classes:

```
type
  EArrayFull = class (Exception);
```

When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

This `Create` constructor (inherited from the `Exception` class) has a string parameter to describe the exception to the user. You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

The code presented in the previous excerpts is part of a sample program called Exception1. Some of the routines have been slightly modified, as in the following `DivideTwicePlusOne` function:

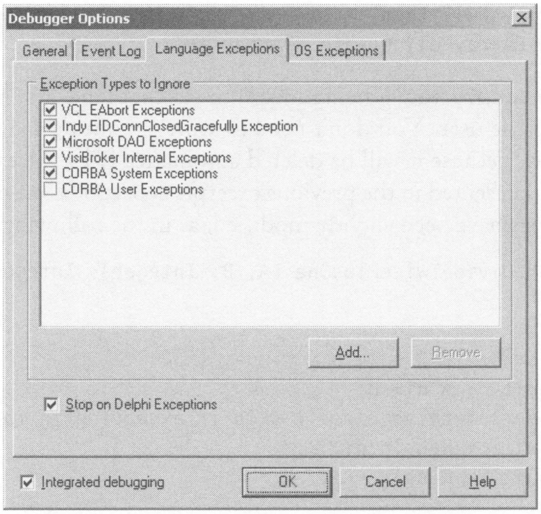
```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero corrected.', mtError, [mbOK], 0);
    end;
    on E: Exception do
    begin
      Result := 0;
      MessageDlg (E.Message, mtError, [mbOK], 0);
    end;
  end; // end except
end;
```

DEBUGGING AND EXCEPTIONS

When you start a program from the Delphi environment (for example, by pressing the F9 key), you'll generally run it within the debugger. When an exception is encountered, the debugger will suspend the program by default. This result is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step by step. You can also use Delphi's Stack Trace feature to see the sequence of function and method calls that caused the program to raise an exception.

In the case of the Exception1 test program, however, this behavior will confuse a programmer not well aware of how Delphi's debugger works. Even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step by step through the code, you can see how it is handled.

If you just want to let the program run when the exception is properly handled, run the program from Windows Explorer, or temporarily disable the Stop on Delphi Exceptions options in the Language Exceptions page of the Debugger Options dialog box (activated by the Tools ➢ Debugger Options command), shown in the Language Exceptions page of the Debugger Options dialog box shown here (as an alternative you can also disable the debugger):



Exception Types to Ignore

☑ VCL EAbort Exceptions
☑ Indy EIDConnClosedGracefully Exception
☑ Microsoft DAO Exceptions
☑ VisiBroker Internal Exceptions
☑ CORBA System Exceptions
☐ CORBA User Exceptions

☑ Stop on Delphi Exceptions

☑ Integrated debugging

In the Exception1 code, there are two different exception handlers after the same `try` block. You can have any number of these handlers, which are evaluated in sequence.

Using a hierarchy of exceptions, a handler is also called for the inherited classes of the type it refers to, as any procedure will do. For this reason, you need to place the broader handlers (the handlers of the ancestor `Exception` classes) at the end. But keep in mind that using a handler for every exception, such as the previous one, is not usually a good choice. It is better to leave unknown exceptions to Delphi. The default exception handler in the VCL displays the error message of the exception class in a message box, and then resumes normal program operation. You can modify the normal exception handler with the `Application.OnException` event or the `OnException` event of the ApplicationEvents component, as demonstrated in the ErrorLog example in the next section.

Another important element of the previous code is the use of the exception object in the handler (see `on E: Exception do`). The reference `E` of class `Exception` refers to the exception object passed by the `raise` statement. When you work with exceptions, remember this rule: You raise an exception by creating an object and handle it by indicating its type. This has an important benefit, because as you have seen, when you handle a type of exception, you are really handling exceptions of the type you specify as well as any descendant type.

Logging Errors

Most of the time, you don't know which operation will raise an exception, and you cannot (and should not) wrap each and every piece of code in a `try/except` block. The general approach is to let Delphi handle all the exceptions and eventually pass them to you, by handling the `OnException` event of the global `Application` object. You can do so rather easily with the ApplicationEvents component.

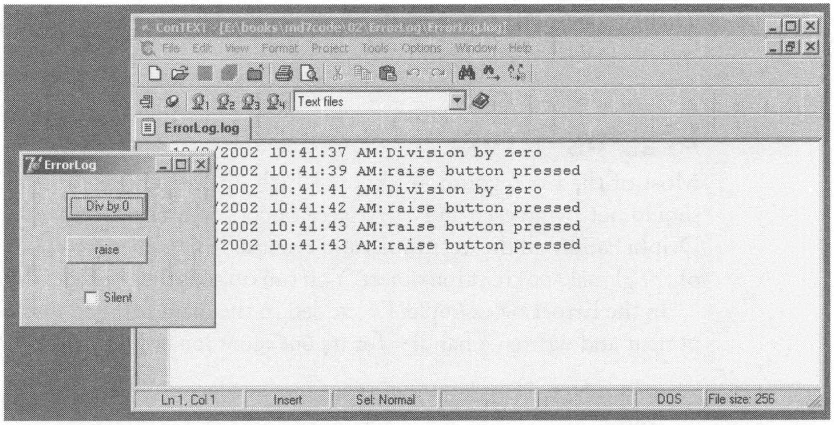
In the ErrorLog example, I've added to the main form an instance of the ApplicationEvents component and written a handler for its `OnException` event:

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);
var
  Filename: string;
  LogFile: TextFile;
begin
  // prepares log file
  Filename := ChangeFileExt (Application.Exename, '.log');
  AssignFile (LogFile, Filename);
  if FileExists (FileName) then
    Append (LogFile) // open existing file
  else
    Rewrite (LogFile); // create a new one
  try
    // write to the file and show error
    Writeln (LogFile, DateTimeToStr (Now) + ':' + E.Message);
    if not CheckBoxSilent.Checked then
      Application.ShowException (E);
  finally
    // close the file
    CloseFile (LogFile);
  end;
end;
```

NOTE *The ErrorLog example uses the text file support provided by the traditional Turbo Pascal TextFile data type. You can assign a text file variable to an actual file and then read or write it. You can find more on TextFile operations in Chapter 12 of the e-book* Essential Pascal, *covered in Appendix C.*

In the global exceptions handler, you can write to the log, for example, the date and time of the event, and also decide whether to show the exception as Delphi usually does (executing the `ShowException` method of the `TApplication` class). By default, Delphi executes `ShowException` only if no `OnException` handler is installed. In Figure 2.8, you can see the ErrorLog program running and a sample exceptions log open in ConTEXT (a nice programmer's editor built with Delphi and available at `www.fixedsys.com/context`).

FIGURE 2.8
The ErrorLog example and the log it produces


ConTEXT - [E:\books\md7code\02\ErrorLog\ErrorLog.log]
File Edit View Format Project Tools Options Window Help
Text files
ErrorLog.log
ErrorLog
Div by 0
raise
☐ Silent
Ln 1, Col 1
Insert
Sel: Normal
DOS
File size: 256

```
/2002 10:41:37 AM:Division by zero
/2002 10:41:39 AM:raise button pressed
/2002 10:41:41 AM:Division by zero
/2002 10:41:42 AM:raise button pressed
/2002 10:41:43 AM:raise button pressed
/2002 10:41:43 AM:raise button pressed
```

Class References

The last language feature I'll discuss in this chapter is the use of *class references*, which implies the idea of manipulating classes themselves within your code. The first point to keep in mind is that a class reference isn't an object; it is a reference to a class type. A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code will make this concept a little clearer.

Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AnObject: TMyClass;
  AClassRef: TMyClassRef;
begin
  AnObject := TMyClass.Create;
  AClassRef := TMyClass;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. There are not many such expressions, but the few cases are interesting, like the creation of an object. You can rewrite the last line of the previous code as follows:

```
AnObject := AClassRef.Create;
```

This time, you apply the `Create` constructor to the class reference instead of to an actual class; you use a class reference to create an object of that class.

Class reference types wouldn't be as useful if they didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef`, you can then assign to it that specific class and any inherited class. So if `TMyNewClass` is an inherited class of `TMyClass` my class, you can also write

```
AClassRef := TMyNewClass;
```

Delphi declares a lot of class references in the run-time library and the VCL, such as the following:

```
TClass = class of TObject;
TComponentClass = class of TComponent;
TFormClass = class of TForm;
```

In particular, the `TClass` class reference type can be used to store a reference to any class you write in Delphi, because every class is ultimately derived from `TObject`. The `TFormClass` reference is used in the source code of most Delphi projects. The `CreateForm` method of the `Application` object requires as a parameter the class of the form to create:

```
Application.CreateForm(TForm1, Form1);
```

The first parameter is a class reference; the second is a variable that stores a reference to the created object instance.

Finally, when you have a class reference, you can apply to it the class methods of the related class. Considering that each class inherits from `TObject`, you can apply to each class reference some of the methods of `TObject`, as you'll see in Chapter 3.

Creating Components Using Class References

What is the *practical* use of class references in Delphi? Being able to manipulate a data type at run time is a fundamental element of the Delphi environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what Delphi does for you behind the scenes.) In other words, class references give you polymorphism for object construction.

To give you a better idea of how class references work, I've built an example named ClassRef. The form displayed by this example has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, push buttons, and edit boxes.

To make this program run properly, you need to change the names of the three components. The form must also have a class reference field, declared as `ClassRef: TControlClass`. It stores a new data type every time the user clicks one of the three radio buttons, with assignments like `ClassRef := TEdit`. The interesting part of the code is executed when the user clicks the form. Again, I've chosen the `OnMouseDown` event of the form to hold the position of the mouse click:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
var
  NewCtrl: TControl;
  MyName: String;
begin
  // create the control
  NewCtrl := ClassRef.Create (Self);
  // hide it temporarily, to avoid flickering
  NewCtrl.Visible := False;
  // set parent and position
  NewCtrl.Parent := Self;
  NewCtrl.Left := X;
  NewCtrl.Top := Y;
  // compute the unique name (and caption)
  Inc (Counter);
  MyName := ClassRef.ClassName + IntToStr (Counter);
  Delete (MyName, 1, 1);
  NewCtrl.Name := MyName;
  // now show it
  NewCtrl.Visible := True;
end;
```

The first line of the code for this method is the key. It creates a new object of the class data type stored in the `ClassRef` field. You accomplish this by applying the `Create` constructor to the class reference. Now you can set the value of the `Parent` property, set the position of the new component, give the component a name (which is automatically used also as the value of `Caption` or `Text`), and make it visible. You can see an example of the output of this program in Figure 2.9.

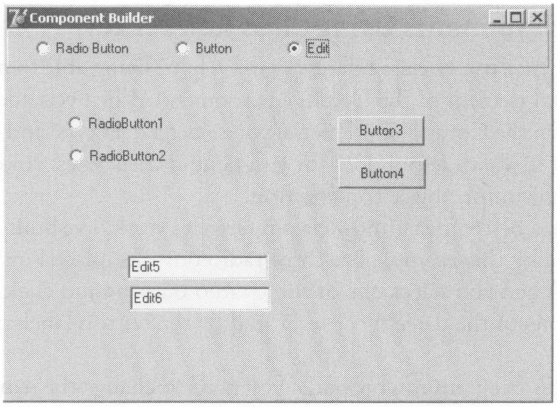


FIGURE 2.9
An example of the output of the ClassRef example

NOTE *For polymorphic construction to work, the base class type of the class reference must have a virtual constructor. If you use a virtual constructor (as in the example), the constructor call applied to the class reference will call the constructor of the type that the class reference variable* currently refers to. *But without a virtual constructor, your code will call the constructor of* fixed class type *indicated in the class reference declaration. Virtual constructors are required for polymorphic construction in the same way that virtual methods are required for polymorphism.*

What's Next?

In this chapter, we have discussed the foundations of object-oriented programming (OOP) in Delphi. We have considered the definition of classes and the use of methods, encapsulation, and memory management, but also some more advanced concepts such as properties and the dynamic creation of components. Then we moved to inheritance, virtual and abstract methods, polymorphism, safe type-casting, interfaces, exceptions, and class references.

This is certainly a lot of information if you are a newcomer. But if you are fluent in another OOP language or if you've already used past versions of Delphi, you should be able to apply the topics covered in this chapter to your programming.

Understanding the secrets of Delphi's language and library is vital for becoming an expert Delphi programmer. These topics form the foundation of working with the VCL and CLX class libraries; after exploring them in the next two chapters, we'll *finally* go on to explore the development of real applications using all the various components provided by Delphi.

In the meantime, Chapter 3 will give you an overview of the Delphi run-time library (mainly a collection of functions with little OOP involved). The RTL is a collection of assorted routines for performing basic tasks with Delphi. Chapter 4 will give you more information about the language, discussing features related to the structure of the Delphi class library, such as the effect of the `published` keyword and the role of events. That chapter, as a whole, will discuss the overall architecture of the component library.

Chapter 3

The Run-Time Library

THE DELPHI PROGRAMMING LANGUAGE favors an object-oriented approach, tied with a visual development style. This is where Delphi shines, and we will cover component-based and visual development in this book; however, I want to underline the fact that many of Delphi's ready-to-use features come from its run-time library (RTL). This is a large collection of functions you can use to perform simple tasks, as well as some complex ones, within your Pascal code. (I use "Pascal" here, because the run-time library primarily contains procedures and functions written with the traditional language constructs and not the OOP extensions added to the language by Borland.)

There is a second reason to devote this chapter of the book to the run-time library: Delphi 6 saw a large number of enhancements to this area, and a few more are provided in Delphi 7. New groups of functions are available, functions have been moved to new units, and other elements have changed, creating a few incompatibilities with older code from which you might be porting your projects. So, even if you've used past versions of Delphi and feel confident with the RTL, you should still read at least portions of this chapter.

This chapter covers the following topics:

- Overview of the RTL
- Delphi RTL functions
- The conversion engine
- Dates, strings, and other new RTL units
- The *TObject* class
- Showing class information at run time

The Units of the RTL

In the most recent versions of Delphi, the RTL has a new structure and several new units. Borland added new units because it also added many new functions. In most cases, you'll find the existing functions in the units where they used to be, but the new functions appear in specific units. For example, new functions related to dates are now in the DateUtils unit, but existing date functions have not been moved out of SysUtils in order to avoid incompatibilities with existing code.

The exception to this rule relates to some of the variant support functions, which were moved out of the System unit to avoid unwanted linkage of specific Windows libraries, even in programs that didn't use those features. These variant functions are now part of the Variants unit, described later in the chapter.

WARNING *Some of your Delphi 4 and Delphi 5 code might need to use the Variants unit to recompile. Delphi is smart enough to acknowledge this requirement and auto-include the Variants unit in projects that use the* `Variant` *type, issuing only a warning.*

A little fine-tuning has also been applied to reduce the minimum size of an executable file, which is at times enlarged by the unwanted inclusion of global variables or initialization code.

EXECUTABLE SIZE UNDER THE MICROSCOPE

While touching up the RTL, Borland engineers have been able to trim a little "fat" out of each and every Delphi application. Reducing the minimum program size by a few KB seems odd, given all the bloated applications these days, but it is a good service to developers. In some cases, even a few KB (multiplied by many applications) can reduce size and eventually download time.

As a simple test, I've built the MiniSize program, which is not an attempt to build the smallest possible program, but rather an attempt to build a very small program that does something interesting: It reports the size of its own executable file. All of the example code is as follows:

```
program MiniSize;

uses
  Windows;

{$R *.RES}

var
  nSize: Integer;
  hFile: THandle;
  strSize: String;

begin
  // open the current file and read the size
  hFile := CreateFile (PChar (ParamStr (0)),
    0, FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
  nSize := GetFileSize (hFile, nil);
  CloseHandle (hFile);

  // copy the size to a string and show it
  SetLength (strSize, 20);
```

Continued on next page

EXECUTABLE SIZE UNDER THE MICROSCOPE *(continued)*

```
  Str (nSize, strSize);
  MessageBox (0, PChar (strSize),
    'Mini Program', MB_OK);
end.
```

The program opens its own executable file, after retrieving its name from the first command-line parameter (`ParamStr (0)`), extracts the size, converts it into a string using the simple `Str` function, and shows the result in a message. The program does not have top-level windows. Moreover, I use the `Str` function for the integer-to-string conversion to avoid including SysUtils, which defines all of the more complex formatting routines and would impose a little extra overhead.

If you compile this program with Delphi 5, you obtain an executable size of 18,432 bytes. Delphi 6 reduces this size to only 15,360 bytes, trimming about 3 KB. In Delphi 7, the size is only slightly greater, at 15,872 bytes. By replacing the long string with a short string and modifying the code a little, you can trim the program further, to less than 10 KB. (You'll end up removing the string support routines and also the memory allocator, something possible only in programs using exclusively low-level calls.) You can find both versions in the source code of the example file.

Notice that decisions of this type always imply a few trade-offs. In eliminating the overhead of variants from Delphi applications that don't use them, for example, Borland added a little extra burden to applications that do. The real advantage of this operation, though, is in the reduced memory footprint of Delphi applications that do not use variants, as a result of not having to bring in several megabytes of the Ole2 system libraries.

What is really important, in my opinion, is the size of full-blown Delphi applications based on run-time packages. A simple test with a do-nothing program, the MiniPack example, shows an executable size of 17,408 bytes.

In the following sections you'll find a list of the RTL units in Delphi, including all the units available (with the complete source code) in the `Source\Rtl\Sys` subfolder of the `Delphi` directory and some of those available in the subfolder `Source\Rtl\Common`. This second directory hosts the source code of units that make up the new RTL package, which comprises both the function-based library and the core classes, discussed at the end of this chapter and in Chapter 4 ("Core Library Classes").

NOTE *The original VCL package present up to version 5 of Delphi has been split into the VCL and RTL packages, so that nonvisual applications using run-time packages don't have the overhead of also deploying visual portions of the VCL. This change also helps with Linux compatibility, because the new package is shared between the VCL and CLX libraries. In addition, notice that the package names in Delphi 6 and 7 don't include the version number; when they are compiled, though, the BPL does have the version in its filename, as discussed in more detail in Chapter 10 ("Libraries and Packages").*

I'll give a short overview of the role of each unit and an overview of the groups of functions included. I'll also devote more space to the newer units. I won't provide a detailed list of the functions included, because the online help includes similar reference material. However, I've tried to pick a few interesting or little-known functions, and I will discuss them shortly.

The System and SysInit Units

System is the core unit of the RTL and is automatically included in any compilation (through an automatic and implicit `uses` statement referring to it). If you try adding the unit to the `uses` statement of a program, you'll get the following compile-time error:

```
[Error] Identifier redeclared: System
```

The System unit includes, among other things:

- The `TObject` class, which is the base class of any class defined in the Object Pascal language, including all the classes of the VCL. (This class is discussed later in this chapter.)
- The `IInterface`, `IInvokable`, `IUnknown`, and `IDispatch` interfaces, as well as the simple implementation class `TInterfacedObject`. `IInterface` was added in Delphi 6 to underscore the point that the interface type in the Delphi language definition is in no way dependent on the Windows operating system. `IInvokable` was added in Delphi 6 to support SOAP-based invocation.
- Some variant support code, including the variant type constants, the `TVarData` record type and the new `TVariantManager` type, a large number of variant conversion routines, and variant record and dynamic array support. This area has seen a lot of changes compared to Delphi 5. The basic information on variants is provided in Chapter 10 of *Essential Pascal* (for more information see Appendix C, "Free Companion Books on Delphi Programming").
- Many base data types, including pointer and array types and the `TDateTime` type described in Chapter 2, "The Delphi Programming Language."
- Memory allocation routines, such as `GetMem` and `FreeMem`, and the actual memory manager, defined by the `TMemoryManager` record and accessed by the `GetMemoryManager` and `SetMemoryManager` functions. For information, the `GetHeapStatus` function returns a `THeapStatus` data structure. Two global variables (`AllocMemCount` and `AllocMemSize`) hold the number and total size of allocated memory blocks. There is more on memory and the use of these functions in Chapter 8, "The Architecture of Delphi Applications" (see in particular the ObjsLeft example).
- Package and module support code, including the `PackageInfo` pointer type, the `GetPackageInfoTable` global function, and the `EnumModules` procedure (package internals are discussed in Chapter 12).
- A rather long list of global variables, including the Windows application instance `MainInstance`; `IsLibrary`, indicating whether the executable file is a library or a stand-alone program; `IsConsole`, indicating console applications; `IsMultiThread`, indicating whether there are secondary threads; and the command-line string `CmdLine`. (The unit also includes `ParamCount` and `ParamStr` for easy access to command-line parameters.) Some of these variables are specific to the Windows platform, some are also available on Linux, and others are specific to Linux.
- Thread-support code, with the `BeginThread` and `EndThread` functions; file support records and file-related routines; wide string and OLE string conversion routines; and many other low-level and system routines (including a number of automatic conversion functions).

The companion unit of System, called SysInit, includes the system initialization code, with functions you'll seldom use directly. This is another unit that is always implicitly included, because it is used by the System unit.

RECENT CHANGES IN THE SYSTEM UNIT

I've already described some interesting features of the System unit in the previous section's list. Most of the changes in recent Delphi versions relate to making the core RTL more cross-platform portable, replacing Windows-specific features with generic implementations now shared by Delphi and Kylix. Along this line, there are new names for interface types, totally revised support for variants, new pointer types, dynamic array support, and functions to customize the management of exception objects.

NOTE *If you read the source code of* `System.pas`*, you'll notice some heavy use of conditional compilation, with many instances of* `{$IFDEF LINUX}` *and* `{$IFDEF MSWINDOWS}` *used to discriminate between the two operating systems. Notice that for Windows, Borland uses the* `MSWINDOWS` *define to indicate the entire platform, because* `WINDOWS` *was used in 16-bit versions of the OS (and contrasts with the symbol* `WIN32`*).*

For example, an addition for compatibility between Linux and Windows relates to line breaks in text files. The `DefaultTextLineBreakStyle` variable affects the behavior of routines that read and write files, including most text-streaming routines. The possible values for this global variable are tlbsLF (the default in Kylix) and tlbsCRLF (the default in Delphi). The line-break style can also be set on a file-by-file basis with `SetTextLineBreakStyle` function.

Similarly, the global `sLineBreak` string constant has the value `#13#10` in the Windows version of the IDE and the value `#10` in the Linux version. Another change is that the System unit now includes the `TFileRec` and `TTextRec` structures, which were defined in the SysUtils unit in earlier versions of Delphi.

The SysUtils and SysConst Units

The SysConst unit defines a few constant strings used by the other RTL units for displaying messages. These strings are declared with the `resourcestring` keyword and saved in the program resources. Like other resources, they can be translated by means of the Integrated Translation Manager or the External Translation Manager.

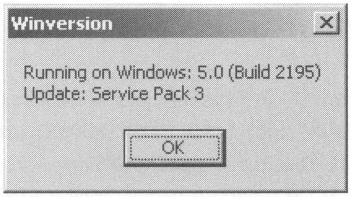
The SysUtils unit is a collection of system utility functions of various types. Unlike other RTL units, it is largely an operating system–dependent unit. The SysUtils unit has no specific focus, but it includes a bit of everything, from string management to locale and multibyte-characters support, from the `Exception` class and several other derived exception classes to a plethora of string-formatting constants and routines. In particular, later in this chapter we'll focus on some of the unit's file management routines.

Some features of SysUtils are used every day by every programmer, such as the `IntToStr` or `Format` string-formatting functions; other features are lesser known, such as the Windows version information global variables. These indicate the Windows platform (Window 9*x* or NT/2000/XP), the operating system version and build number, and the service pack installed. They can be used as in the following code, extracted from the WinVersion example:

```
case Win32Platform of
  VER_PLATFORM_WIN32_WINDOWS:  ShowMessage ('Windows 9x');
```

```
    VER_PLATFORM_WIN32_NT:      ShowMessage ('Windows NT');
  end;

  ShowMessage ('Running on Windows: ' + IntToStr (Win32MajorVersion) + '.' +
    IntToStr (Win32MinorVersion) + ' (Build ' + IntToStr (Win32BuildNumber) +
    ') ' + #10#13 + 'Update: ' + Win32CSDVersion);
```

The second code fragment produces a message like the one in the following graphic (of course, on the operating-system version you have installed):



Running on Windows: 5.0 (Build 2195)
Update: Service Pack 3

Another little-known feature of this unit is the `TMultiReadExclusiveWriteSynchronizer` class—probably the VCL class with the longest name. Borland has defined an alias name for the class, which is much shorter: `TMREWSync` (the two classes are identical). This class supports multithreading: It allows you to work with resources that can be used by multiple threads at the same time for reading (multiread) but must be used by a single thread when writing (exclusive-write). This means writing cannot begin until all the reading threads have terminated.

The implementation of the `TMultiReadExclusiveWriteSynchronizer` class has been updated in Delphi 7, but similar improvements are available in an informal patch released after Delphi 6 update 2. The new version of the class is more optimized and less subject to deadlocks, which are often a problem with synchronization code.

NOTE *The multiread synchronizer is unique in that it supports recursive locks and promotion of read locks to write locks. The main purpose of the class is to allow multiple threads easy, fast access to read from a shared resource, but still allow one thread to gain exclusive control of the resource for relatively infrequent updates. Delphi includes other synchronization classes, declared in the SyncObjs unit (available under* `Source/Rtl/Common`*) and closely mapped to operating-system synchronization objects (such as events and critical sections in Windows).*

RECENT SYSUTILS FUNCTIONS

Over the last couple of versions, Delphi has added some new functions within the SysUtils unit. One of these areas relates to Boolean-to-string conversion. The `BoolToStr` function generally returns '–1' and '0' for true and false values. If the second optional parameter is specified, the function returns the first string in the `TrueBoolStrs` and `FalseBoolStrs` arrays (by default 'TRUE' and 'FALSE'):

```
BoolToStr (True) // returns '-1'
BoolToStr (False, True) // returns 'FALSE' by default
```

The reverse function is `StrToBool`, which can convert a string containing either one of the values of the two Boolean arrays mentioned or a numeric value. In the latter case, the result will be true unless the numeric value is zero. You can see a simple demo of the use of the Boolean conversion functions in the StrDemo example, later in this chapter.

Other functions recently added to SysUtils relate to floating-point conversions to currency and date time types: You can use `FloatToCurr` and `FloatToDateTime` to avoid an explicit typecast. The `TryStrToFloat` and `TryStrToCurr` functions try to convert a string into a floating-point or currency value and will return False in case of error instead of generating an exception (as the classic `StrToFloat` and `StrToCurr` functions do).

The `AnsiDequotedStr` function, which removes quotes from a string, matches the `AnsiQuoteStr` function added in Delphi 5. Speaking of strings, as of Delphi 6 there is much-improved support for wide strings, with a series of routines including `WideUpperCase`, `WideLowerCase`, `WideCompareStr`, `WideSameStr`, `WideCompareText`, `WideSameText`, and `WideFormat`. All of these functions work like their `AnsiString` counterparts.

Three functions (`TryStrToDate`, `TryEncodeDate`, and `TryEncodeTime`) try to convert a string to a date or to encode a date or time, without raising an exception, similar to the `Try` functions previously mentioned. In addition, the `DecodeDateFully` function returns more detailed information, such as the day of the week, and the `CurrentYear` function returns the year of today's date.

A portable, friendly, overloaded version of the `GetEnvironmentVariable` function uses string parameters instead of PChar parameters and is definitely easier to use than the original version based on `PChar` pointers:

```
function GetEnvironmentVariable(Name: string): string;
```

Other functions relate to interface support. Two overloaded versions of the little-known `Support` function allow you to check whether an object or a class supports a given interface. The function corresponds to the behavior of the `is` operator for classes and is mapped to the `QueryInterface` method. Here's an example:

```
var
  W1: IWalker;
  J1: IJumper;
begin
  W1 := TAthlete.Create;
  // more code...
  if Supports (w1, IJumper) then
  begin
    J1 := W1 as IJumper;
    Log (J1.Walk);
  end;
```

SysUtils also includes an `IsEqualGUID` function and two functions for converting strings to GUIDs and vice versa. The function `CreateGUID` has been moved to SysUtils, as well, to make it available on Linux (with a custom implementation, of course).

Finally, more features were added in recent versions to improve cross-platform support. The `AdjustLineBreaks` function can now do different types of *adjustments* to carriage-return and line-feed sequences, and new global variables for text files have been introduced in the System unit, as described earlier. The `FileCreate` function has an overloaded version in which you can specify file-access rights *the Unix way*. The `ExpandFileName` function can locate files (on case-sensitive file systems) even when their cases don't exactly correspond. The functions related to path delimiters (backslash or slash) have been made more generic than in earlier versions of Delphi and renamed accordingly. (For example, the old `IncludeTrailingBackslash` function is now better known as `IncludingTrailingPathDelimiter`.)

Speaking of files, Delphi 7 adds to the SysUtils unit the `GetFileVersion` function, which reads the version number from the version information optionally added to a Windows executable file (which is why this function won't work on Linux).

DELPHI 7 EXTENDED STRING FORMATTING ROUTINES

Most of Delphi's string formatting routines (see Appendix C, "Free Companion Books on Delphi," for instructions on how to get an e-book introducing some of them) use global variables to determine decimal and thousand separators, date/time formats, and so on. The values of these variables are first read from the system (Windows regional settings) when a program starts, and you are free to override each of them. However, if the user modifies the Regional Settings in Control Panel while your program is running, the program will respond to the broadcast message by updating the variables, probably losing your hard-coded changes.

If you need different output formats in different places within a program, you can take advantage of the new set of overloaded string formatting routines; they take an extra parameter of type `TFormatSettings`, including all the relevant settings. For example, there are now two versions of `Format`:

```
function Format(const Format: string;
  const Args: array of const): string; overload;
function Format(const Format: string; const Args: array of const;
  const FormatSettings: TFormatSettings): string; overload;
```

Tens of functions have this new extra parameter, which is then used instead of the global settings. However, you can initialize it with the default settings of the computer on which your program is running by calling the new `GetLocaleFormatSettings` function (available only on Windows, not Linux).

The Math Unit

The Math unit hosts a collection of mathematical functions: about 40 trigonometric functions, logarithmic and exponential functions, rounding functions, polynomial evaluations, almost 30 statistical functions, and a dozen financial functions.

Describing all the functions of this unit would be rather tedious, although some readers are probably very interested in Delphi's mathematical capabilities. For this reason, I've decided to focus on math functions introduced in the latest versions of Delphi (particularly Delphi 6) and then cover one specific topic that often confuses Delphi programmers—rounding.

NEW MATH FUNCTIONS

Recent versions add to the Math unit quite a number of features. They include support for infinite constants (`Infinity` and `NegInfinity`) and related comparison functions (`IsInfinite` and `IsNan`), along with new trigonometric functions for cosecants and cotangents, and new angle-conversion functions.

A handy feature is the availability of an overloaded `IfThen` function, which returns one of two possible values depending on a Boolean expression. (A similar function is available also for strings.) You can use it, for example, to compute the minimum of two values:

```
nMin := IfThen (nA < nB, na, nB);
```

NOTE *The `IfThen` function is similar to the `?:` operator of the C/C++ language. I find it handy because you can replace a complete `if/then/else` statement with a much shorter expression, writing less code and often declaring fewer temporary variables.*

You can use `RandomRange` and `RandomFrom` instead of the traditional `Random` function to gain more control over the random values produced by the RTL. The first function returns a number within two extremes you specify, and the second selects a random value from an array of possible numbers you pass to it as a parameter.

The `InRange` Boolean function can be used to check whether a number is within two other values. The `EnsureRange` function, on the other hand, forces the value to be within the specified range. The return value is the number itself or the lower limit or upper limit, in the event the number is out of range. Here is an example:

```
// do something only if value is within min and max
if InRange (value, min, max) then
  ...

// make sure the value is between min and max
value := EnsureRange (value, min, max);
...
```

Another set of useful functions relates to comparisons. Floating-point numbers are fundamentally inexact; a floating-point number is an approximation of a theoretical real value. When you do mathematical operations on floating-point numbers, the inexactness of the original values accumulates in the results. Multiplying and dividing by the same number might not return exactly the original number but one that is very close to it. The `SameValue` function allows you to check whether two values are close enough in value to be considered equal. You can specify how close the two numbers should be or let Delphi compute a reasonable error range for the representation you are using. (This is why the function is overloaded.) Similarly, the `IsZero` function compares a number to zero, with the same "fuzzy logic."

The `CompareValue` function uses the same rule for floating-point numbers but is available also for integers; it returns one of the three constants `LessThanValue`, `EqualsValue`, and `GreaterThanValue` (corresponding to –1, 0, and 1). Similarly, the new `Sign` function returns –1, 0, or 1 to indicate a negative value, zero, or a positive value.

The `DivMod` function is equivalent to both the div and mod operations, returning the result of the integer division and the remainder (or modulus) at once. The `RoundTo` function allows you to specify the rounding digit—allowing, for example, rounding to the nearest thousand or to two decimals:

```
RoundTo (123827, 3);   // result is 124,000
RoundTo (12.3827, -2); // result is 12.38
```

WARNING *Notice that the `RoundTo` function uses a positive number to indicate the power of 10 to round to (for example, 2 for hundreds) or a negative number for the number of decimal places. This is exactly the opposite of the `Round` function used by spreadsheets such as Excel.*

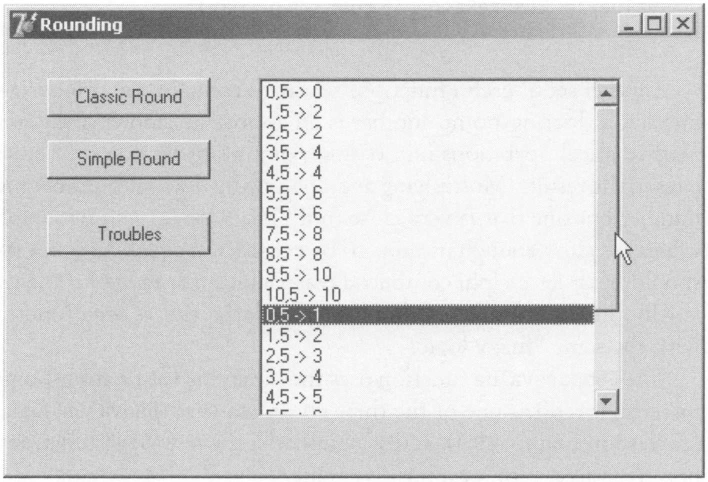
There have also been some changes to the standard rounding operations provided by the `Round` function: You can now control how the FPU (the floating-point unit of the CPU) does the rounding by calling the `SetRoundMode` function. Other functions control the FPU precision mode and its exceptions.

ROUNDING HEADACHES

Delphi's classic `Round` function and the newer `RoundTo` functions are mapped to the CPU/FPU rounding algorithms. By default, Intel CPUs use *banker's rounding,* which is also the type of rounding typically found in spreadsheet applications.

Banker's rounding is based on the assumption that when you're rounding numbers that lie exactly between two values (the .5 numbers), rounding them all up or all down will statistically increase or reduce the total amount (of money, in general). For this reason, the rule of banker's rounding indicates that .5 numbers should be rounded down or up depending on whether the number (without decimals) is odd or even. This way, the rounding will be balanced, at least statistically. You can see an example of the output of banker's rounding in Figure 3.1, which shows the output of the Rounding example I've built to demonstrate different types of rounding.

FIGURE 3.1
The Rounding example, demonstrated banker's rounding and arithmetic rounding



```
0,5 -> 0
1,5 -> 2
2,5 -> 2
3,5 -> 4
4,5 -> 4
5,5 -> 6
6,5 -> 6
7,5 -> 8
8,5 -> 8
9,5 -> 10
10,5 -> 10
0,5 -> 1
1,5 -> 2
2,5 -> 3
3,5 -> 4
4,5 -> 5
```

The program also uses another type of rounding provided by the Math unit in the `SimpleRoundTo` function, which uses *asymmetric arithmetic rounding*. In this case, all .5 numbers are rounded to the upper value. However, as highlighted in the Rounding example, the function doesn't work as expected when rounding to a decimal digit (that is, when you pass a negative second parameter). In this case, due to the representation errors of floating-point numbers, the rounding trims the values; for example, it turns 1.15 into 1.1 instead of the expected 1.2. The solution is to multiply the value by ten before rounding, round it to zero decimal digits, and then divide it, as demonstrated in the sample program:

```
SimpleRoundTo (d * 10, 0) / 10)
```

The ConvUtils and StdConvs Units

The ConvUtils unit contains the core of the conversion engine introduced in Delphi 6. It uses the conversion constants defined by a second unit, StdConvs. I'll cover these two units later in this chapter and show how to extend them with new measurement units.

NOTE *Delphi 7 makes only a single improvement to the conversion unit: It adds support for* stones *(the British unit of measurement that is equivalent to 14 pounds). In any case, if you have to deal with units of measurements in your code, you'll appreciate the features available in this engine.*

The DateUtils Unit

The DateUtils unit is a collection of date- and time-related functions. It includes functions for picking values from a `TDateTime` variable or counting values from a given interval, such as

```
// pick value
function DayOf(const AValue: TDateTime): Word;
function HourOf(const AValue: TDateTime): Word;
// value in range
function WeekOfYear(const AValue: TDateTime): Integer;
function HourOfWeek(const AValue: TDateTime): Integer;
function SecondOfHour(const AValue: TDateTime): Integer;
```

Some of these functions are quite odd, such as `MilliSecondOfMonth` or `SecondOfWeek`, but Borland developers have decided to provide a complete set of functions, no matter how impractical they sound. (I actually used some of these functions in Chapter 2, to build the `TDate` class.)

There are functions for computing the initial or final value of a given time interval (day, week, month, year) including the current date, and for range checking and querying; for example:

```
function DaysBetween(const ANow, AThen: TDateTime): Integer;
function WithinPastDays(const ANow, AThen: TDateTime;
  const ADays: Integer): Boolean;
```

Other functions cover incrementing and decrementing by each of the possible time intervals, encoding and "recoding" (replacing one element of the `TDateTime` value, such as the day, with a new one), and doing "fuzzy" comparisons (approximate comparisons where a difference of a millisecond will still make two dates equal). Overall, DateUtils is quite interesting and not terribly difficult to use.

The StrUtils Unit

The StrUtils unit was introduced in Delphi 6 with some new string-related functions. One of the key features of this unit is the availability of many string comparison functions. There are functions based on a *soundex* algorithm (`AnsiResembleText`), and others that provide lookup in arrays of strings (`AnsiMatchText` and `AnsiIndexText`), substring location, and text replacement (including `AnsiContainsText` and `AnsiReplaceText`).

NOTE Soundex *is an algorithm that compares names based on how they sound rather than how they are spelled. The algorithm computes a number for each word sound, so that by comparing two such numbers you can determine whether two names sound similar. The system was first applied in 1880 by the U.S. Bureau of the Census; it was patented in 1918 and is now in the public domain. The soundex code is an indexing system that translates a name into a four-character code consisting of one letter and three numbers. More information is available at* `www.nara.gov/genealogy/coding.html`.

Beside comparisons, other functions provide a two-way test (the nice `IfThen` function, similar to the one we've already seen for numbers), duplicate and reverse strings, and replace substrings. Most of these string functions were added as a convenience to Visual Basic programmers migrating to Delphi.

I've used some of these functions in the StrDemo example, which uses also some of the Boolean-to-string conversions defined within the SysUtils unit. The program is little more than a test for a few of these functions. For example, it uses the soundex comparison between the strings entered in two edit boxes, converting the resulting Boolean into a string and showing it:

```
ShowMessage (BoolToStr (AnsiResemblesText
  (EditResemble1.Text, EditResemble2.Text), True));
```

The program also showcases the `AnsiMatchText` and `AnsiIndexText` functions, after filling a dynamic array of strings (called `strArray`) with the values of the strings inside a list box. I could have used the simpler `IndexOf` method of the `TStrings` class, but doing so would have defeated the purpose of the example. The two list comparisons are as follows:

```
procedure TForm1.ButtonMatchesClick(Sender: TObject);
begin
  ShowMessage (BoolToStr (AnsiMatchText(EditMatch.Text, strArray), True));
end;

procedure TForm1.ButtonIndexClick(Sender: TObject);
var
  nMatch: Integer;
begin
  nMatch := AnsiIndexText(EditMatch.Text, strArray);
  ShowMessage (IfThen (nMatch >= 0, 'Matches the string number ' +
    IntToStr (nMatch), 'No match'));
end;
```

Notice the use of the `IfThen` function in the last few lines of code; it has two alternative output strings, depending on the result of the initial test (`nMatch >= 0`).

Three more buttons do simple calls to three other new functions, with the following lines of code (one for each):

```
// duplicate (3 times) a string
ShowMessage (DupeString (EditSample.Text, 3));
// reverse the string
ShowMessage (ReverseString (EditSample.Text));
// choose a random string
ShowMessage (RandomFrom (strArray));
```

FROM POS TO POSEX

Delphi 7 adds a little to the StrUtils unit. The new `PosEx` function will be handy to many developers and is worth a brief mention. When searching for multiple occurrences of a string within another one, a classic Delphi solution was to use the `Pos` function and repeat the search over the remaining portion of the string. For example, you could count the occurrences of a string inside another string with code like this:

```
function CountSubstr (text, sub: string): Integer;
var
  nPos: Integer;
begin
  Result := 0;
  nPos := Pos (sub, text);
  while nPos > 0 do
  begin
    Inc (Result);
    text := Copy (text, nPos + Length (sub), MaxInt);
    nPos := Pos (sub, text);
  end;
end;
```

The new `PosEx` function allows you to specify the starting position of the search within a string, so you don't need to alter the original string (wasting quite some time). Thus the previous code can be simplified in the following way:

```
function CountSubstrEx (text, sub: string): Integer;
var
  nPos: Integer;
begin
  Result := 0;
  nPos := PosEx (sub, text, 1); // default
  while nPos > 0 do
  begin
    Inc (Result);
    nPos := PosEx (sub, text, nPos + Length (sub));
  end;
end;
```

Both code snippets are used in a trivial way in the StrDemo example discussed earlier.

The Types Unit

The Types unit holds data types common to multiple operating systems. In past versions of Delphi, the same types were defined by the Windows unit; now they've been moved to this common unit, shared by Delphi and Kylix. The types defined here are simple and include, among others, the `TPoint`, `TRect`, and `TSmallPoint` record structures plus their related pointer types.

WARNING *Notice that you will have to update old Delphi programs that refer to* `TRect` *or* `TPoint`*, by adding the Types unit in the* `uses` *statement; otherwise they won't compile.*

The Variants and VarUtils Units

Variants and VarUtils are two more units introduced in Delphi 6 to host the variant-related portion of the library. The Variants unit contains generic code for variants. As mentioned earlier, some of the routines in this unit have been moved here from the System unit. Functions include generic variant support, variant arrays, variant copying, and dynamic array to variant array conversions. In addition, the `TCustomVariantType` class defines customizable variant data types.

The Variants unit is totally platform independent and uses the VarUtils unit, which contains OS-dependent code. In Delphi, this unit uses the system APIs to manipulate variant data; in Kylix, it uses custom code provided by the RTL library.

NOTE *In Delphi 7, these units have been extended and some bugs have been patched. The variant implementation has been heavily reworked behind the scenes to improve the speed of this technology and decrease the memory footprint of its code.*

A specific area that has seen significant improvement in Delphi 7 is the ability to control the behavior of variant implementations, particularly comparison rules. Delphi 6 saw a change in the variant code so that *null* values cannot be compared with other values. This behavior is correct from a formal point of view, specifically for the fields of a dataset (an area in which variants are heavily used), but this change had the side effect of breaking existing code. Now you can control this behavior using the `NullEqualityRule` and `NullMagnitudeRule` global variables, each of which assumes one of the following values:

ncrError Any type of comparison causes an exception to be raised, because you cannot compare an undefined value; this was the (new) default behavior in Delphi 6.

ncrStrict Any type of comparison always fails (returning False), regardless of the values.

ncrLoose Equality tests succeed only among null values (a null is different from any other value). In comparisons null values are considered like empty values or zeros.

Other settings like `NullStrictConvert` and `NullAsStringValue` control how conversion is accomplished in case of null values. I suggest that you carry out your own experiments using the VariantComp example available with the code for this chapter. As you can see in Figure 3.2, this program has a form with a RadioGroup you can use to change the settings of the `NullEqualityRule` and `NullMagnitudeRule` global variables, and a few buttons to perform various comparisons.

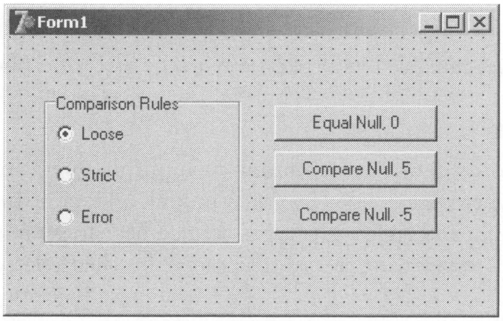


FIGURE 3.2
The form of the VariantComp example at design time

CUSTOM VARIANTS AND COMPLEX NUMBERS

Another recent extension to the concept of variants is the possibility of extending the type system with custom variants. This technique allows you to define a new data type that, contrary to a class, overloads standard arithmetic operators.

A variant is a type holding both the type specification and the actual value. One variant can contain a string; another can contain a number. The system defines automatic conversions among variant types, allowing you to mix them inside operations (including custom variants). This flexibility comes at a high cost: Operations on variants are much slower than on native types, and variants use extra memory.

As an example of a custom variant type, Delphi ships with an interesting definition for complex numbers, found in the VarCmplx unit (available in source-code format in the `Rtl\Common` folder). You can create complex variants by using one of the overloaded `VarComplexCreate` functions and use them in any expression, as the following code fragment demonstrates:

```
var
  v1, v2: Variant;
begin
  v1 := VarComplexCreate (10, 12);
  v2 := VarComplexCreate (10, 1);
  ShowMessage (v1 + v2 + 5);
```

The complex numbers are defined using classes, but they are surfaced as variants by inheriting a new class from the `TCustomVariantType` class (defined in the Variants unit), overriding a few virtual abstract functions, and creating a global object that takes care of the registration within the system.

Besides these internal definitions, the Variants unit includes a long list of routines for operating on variants, including mathematical and trigonometric operations. I'll leave them to your study, because not all readers will be interested in complex numbers for their programs.

WARNING *Building a custom variant is certainly not an easy task, and I can hardly find reasons for using them instead of objects and classes. With a custom variant you gain the advantage of using operator overloading on your own data structures, but you lose compile-time checking, make the code much slower, miss several OOP features, and have to write a lot of rather complex code.*

The DelphiMM and ShareMem Units

The DelphiMM and ShareMem units relate to memory management. The standard Delphi memory manager is declared in the System unit.

The DelphiMM unit defines an alternative memory manager library to be used when passing strings from an executable to a DLL (a Windows dynamic linking library), both built with Delphi. This memory manager library is compiled by default in the `Borlndmm.dll` library file you'll have to deploy with your program.

The interface to this memory manager is defined in the ShareMem unit. You must include this unit (it's required to be the first unit) in the projects of both your executable and library, as described in more detail in Chapter 10, "Libraries and Packages."

NOTE *Unlike Delphi, Kylix has no DelphiMM and ShareMem units, because memory management is provided in the native Linux libraries (in particular, Kylix uses* `malloc` *from glibc) and so is effectively shared among different modules. In Kylix, however, applications with multiple modules must use the ShareExcept unit, which allows exceptions raised in a module to be surfaced to another module.*

COM-Related Units

ComConst, ComObj, and ComServ provide low-level COM support. These units are not really part of the RTL, from my point of view, so I won't discuss them here in any detail. You can refer to Chapter 12 for all the related information. These units have not changed much in recent versions of Delphi.

Converting Data

As mentioned earlier in this chapter, Delphi includes a new conversion engine, defined in the Conv Utils unit. The engine by itself doesn't include any definition of actual measurement units; instead, it has a series of core functions for end users.

The key function is the conversion call, the `Convert` function. You simply provide the amount, the units it is expressed in, and the units you want it converted into. The following converts a temperature of 31 degrees Celsius to Fahrenheit:

```
Convert (31, tuCelsius, tuFahrenheit)
```

An overloaded version of the `Convert` function lets you convert values that have two units, such as speed (which has both a length unit and a time unit). For example, you can convert miles per hour to meters per second with this call:

```
Convert (20, duMiles, tuHours, duMeters, tuSeconds)
```

Other functions in the unit allow you to convert the result of an addition or a difference, check if conversions are applicable, and even list the available conversion families and units.

A predefined set of measurement units is provided in the StdConvs unit. This unit has conversion families and an impressive number of values, as shown in the following reduced excerpt:

```
// Distance Conversion Units
// basic unit of measurement is meters
cbDistance: TConvFamily;

duAngstroms: TConvType;
duMicrons: TConvType;
duMillimeters: TConvType;
duMeters: TConvType;
duKilometers: TConvType;
duInches: TConvType;
duMiles: TConvType;
duLightYears: TConvType;
duFurlongs: TConvType;
duHands: TConvType;
duPicas: TConvType;
```

This family and the various units are registered in the conversion engine in the initialization section of the unit, providing the conversion ratios (saved in a series of constants, such as `MetersPerInch` in the following code):

```
cbDistance := RegisterConversionFamily('Distance');
duAngstroms := RegisterConversionType(cbDistance, 'Angstroms', 1E-10);
duMillimeters := RegisterConversionType(cbDistance, 'Millimeters', 0.001);
duInches := RegisterConversionType(cbDistance, 'Inches', MetersPerInch);
```

To test the conversion engine, I built a generic example (ConvDemo) that allows you to work with the entire set of available conversions. The program fills a combo box with the available conversion families and a list box with the available units of the active family. This is the code:

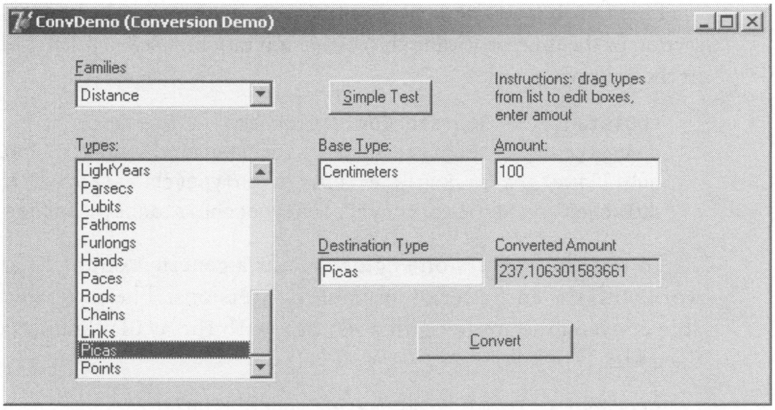
```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  GetConvFamilies (aFamilies);
  for i := Low(aFamilies) to High(aFamilies) do
    ComboFamilies.Items.Add (ConvFamilyToDescription (aFamilies[i]));
  // get the first and fire event
  ComboFamilies.ItemIndex := 0;
  ChangeFamily (self);
end;

procedure TForm1.ChangeFamily(Sender: TObject);
var
  aTypes: TConvTypeArray;
  i: Integer;
begin
  ListTypes.Clear;
  CurrFamily := aFamilies [ComboFamilies.ItemIndex];
  GetConvTypes (CurrFamily, aTypes);
  for i := Low(aTypes) to High(aTypes) do
    ListTypes.Items.Add (ConvTypeToDescription (aTypes[i]));
end;
```

The `aFamilies` and `CurrFamily` variables are declared in the private section of the form as follows:

```
aFamilies: TConvFamilyArray;
CurrFamily: TConvFamily;
```

At this point, a user can enter two measurement units and an amount in the corresponding edit boxes on the form, as you can see in Figure 3.3. To make the operation faster, the user can select a value in the list and drag it to one of the two Type edit boxes. The dragging support is described in the following sidebar "Simple Dragging in Delphi."

FIGURE 3.3
The ConvDemo example at run time



Families: Distance

Instructions: drag types from list to edit boxes, enter amout

Base Type: Centimeters

Amount: 100

Destination Type: Picas

Converted Amount: 237,106301583661

SIMPLE DRAGGING IN DELPHI

The ConvDemo example, built to show how to use the conversion engine, uses an interesting technique: dragging. You can move the mouse over the list box, select an item, and then keep the left mouse button pressed and drag the item over one of the edit boxes in the center of the form.

To accomplish this functionality, I had to set the `DragMode` property of the list box (the source component) to dmAutomatic and implement the `OnDragOver` and `OnDragDrop` events of the target edit boxes (the two edit boxes are connected to the same event handlers, sharing the same code). In the first method, the program indicates that the edit boxes always accept the dragging operation, regardless of the source. In the second method, the program copies the text selected in the list box (the `Source` control of the dragging operation) to the edit box that fired the event (the `Sender` object). Here is the code for the two methods:

```
procedure TForm1.EditTypeDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
   Accept := True;
end;

procedure TForm1.EditTypeDragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  (Sender as TEdit).Text := (Source as TListBox).Items
    [(Source as TListBox).ItemIndex];
end;
```

The units must match those available in the current family. In case of error, the text in the Type edit boxes is shown in red. This is the effect of the first part of the form's `DoConvert` method, which is activated as soon as the value of one of the edit boxes for the units or the amount changes. After checking the types in the edit boxes, the `DoConvert` method performs the conversion, displaying the result in the fourth, grayed edit box. In case of errors, you'll get an appropriate message in the same box. Here is the code:

```
procedure TForm1.DoConvert(Sender: TObject);
var
  BaseType, DestType: TConvType;
begin
  // get and check base type
  if not DescriptionToConvType(CurrFamily, EditType.Text, BaseType) then
    EditType.Font.Color := clRed
  else
    EditType.Font.Color := clBlack;

  // get and check destination type
  if not DescriptionToConvType(CurrFamily, EditDestination.Text,
      DestType) then
    EditDestination.Font.Color := clRed
  else
    EditDestination.Font.Color := clBlack;

  if (DestType = 0) or (BaseType = 0) then
    EditConverted.Text := 'Invalid type'
  else
    EditConverted.Text := FloatToStr (Convert (
      StrToFloat (EditAmount.Text), BaseType, DestType));
end;
```

If all this is not interesting enough for you, consider that the conversion types provided serve only as a demo: You can fully customize the engine by providing the measurement units you are interested in, as described in the next section.

What About Currency Conversions?

Converting currencies is not exactly the same as converting measurement units, because currency rates change constantly. In theory, you can register a conversion rate with Delphi's conversion engine. From time to time, you check the new rate of exchange, unregister the existing conversion, and register a new one. However, keeping up with the actual rate means changing the conversion so often that the operation might not make a lot of sense. Also, you'll have to triangulate conversions: You must define a base unit (probably the U.S. dollar if you live in America) and convert to and from this currency even if you're converting between two different currencies.

It's more interesting to use the engine to convert member currencies of the euro, for two reasons. First, conversion rates are fixed (until the single euro currency takes over). Second, the conversion among euro currencies is legally done by converting a currency to euros first and then from the euro amount to the other currency—the exact behavior of Delphi's conversion engine. There is one small problem: You should apply a rounding algorithm at every step of the conversion. I'll consider this problem after I've provided the base code for integrating euro currencies with the Delphi conversion engine.

NOTE *The ConvertIt demo available among the Delphi examples provides support for euro conversions, using a slightly different rounding approach, still not as precise as demanded by the European currency conversion rules. I've decided to keep this example, because it is instructive in showing how to create a new measurement system.*

The example, called EuroConv, teaches how to register any new measurement unit with the engine. Following the template provided by the StdConvs unit, I've created a new unit (called EuroConvConst). In the interface portion, I've declared variables for the family and the specific units, as follows:

```
interface

var
  // Euro Currency Conversion Units
  cbEuroCurrency: TConvFamily;

  cuEUR: TConvType;
  cuDEM: TConvType; // Germany
  cuESP: TConvType; // Spain
  cuFRF: TConvType; // France
  // and so on...
```

The implementation portion of the unit defines constants for the various official conversion rates:

```
implementation

const
  DEMPerEuros = 1.95583;
  ESPPerEuros = 166.386;
  FRFPerEuros = 6.55957;
  // and so on...
```

Finally, the unit initialization code registers the family and the various currencies, each with its own conversion rate and a readable name:

```
initialization
  // Euro Currency's family type
  cbEuroCurrency := RegisterConversionFamily('EuroCurrency');

  cuEUR := RegisterConversionType(
    cbEuroCurrency, 'EUR', 1);
  cuDEM := RegisterConversionType(
    cbEuroCurrency, 'DEM', 1 / DEMPerEuros);
  cuESP := RegisterConversionType(
    cbEuroCurrency, 'ESP', 1 / ESPPerEuros);
  cuFRF := RegisterConversionType(
    cbEuroCurrency, 'FRF', 1 / FRFPerEuros);
```

NOTE *The engine uses as a conversion factor the amount of the base unit necessary to obtain the secondary units, with a constant like* `MetersPerInch`*, for example. The standard rate of euro currencies is defined the opposite way. For this reason, I've kept the conversion constants with the official values (like* `DEMPerEuros`*) and passed them to the engine as fractions (*`1/DEMPerEuros`*).*

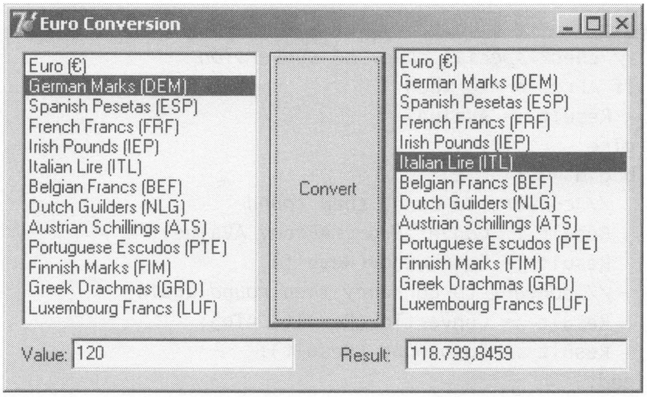
Having registered this unit, you can now convert 120 German marks to Italian liras as follows:

```
Convert (120, cuDEM, cuITL)
```

The demo program does a little more: It provides two list boxes with the available currencies, extracted as in the previous example, and edit boxes for the input value and final result. You can see the form at run time in Figure 3.4.

FIGURE 3.4

The output of the EuroConv unit, showing the use of Delphi's conversion engine with a custom measurement unit


Euro Conversion
Euro (€)
German Marks (DEM)
Spanish Pesetas (ESP)
French Francs (FRF)
Irish Pounds (IEP)
Italian Lire (ITL)
Belgian Francs (BEF)
Dutch Guilders (NLG)
Austrian Schillings (ATS)
Portuguese Escudos (PTE)
Finnish Marks (FIM)
Greek Drachmas (GRD)
Luxembourg Francs (LUF)
Convert
Euro (€)
German Marks (DEM)
Spanish Pesetas (ESP)
French Francs (FRF)
Irish Pounds (IEP)
Italian Lire (ITL)
Belgian Francs (BEF)
Dutch Guilders (NLG)
Austrian Schillings (ATS)
Portuguese Escudos (PTE)
Finnish Marks (FIM)
Greek Drachmas (GRD)
Luxembourg Francs (LUF)

Value: 120

Result: 118.799,8459

The program works nicely but is not perfect, because the proper rounding is not applied; you should round not only the final result of the conversion but also the intermediate value. Using the conversion engine to accomplish this rounding directly is not easy. The engine allows you to provide either a custom conversion function or a conversion rate. But writing identical conversion functions for all the currencies seems like a bad idea, so I've decided to take a different path. (You can see examples of custom conversion functions in the StdConvs unit, in the portion related to temperatures.)

In the EuroConv example, I've added to the unit with the conversion rates a custom `EuroConv` function that does the proper conversion. Simply calling this function instead of the standard `Convert` function does the trick (and I see no drawback to this approach, because in such programs you'll rarely mix currencies with distances or temperatures). As an alternative, I could have inherited a new class from `TConvTypeFactor`, providing a new version of the `FromCommon` and `ToCommon` methods; or I could have called the overloaded version of the `RegisterConversionType` that accepts these two functions as parameters. However, neither of these techniques would have allowed me to handle special cases, such as the conversion of a currency to itself.

This is the code of the `EuroConv` function, which uses the internal `EuroRound` function to round to the number of digits specified in the `Decimals` parameter (which must be between 3 and 6, according to the official rules):

```
type
  TEuroDecimals = 3..6;

function EuroConvert (const AValue: Double;
  const AFrom, ATo: TConvType;
  const Decimals: TEuroDecimals = 3): Double;

  function EuroRound (const AValue: Double): Double;
  begin
    Result := AValue * Power (10, Decimals);
    Result := Round (Result);
    Result := Result / Power (10, Decimals);
  end;

begin
  // check special case: no conversion
  if AFrom = ATo then
    Result := AValue
  else
  begin
    // convert to Euro, then round
    Result := ConvertFrom (AFrom, AValue);
    Result := EuroRound (Result);
    // convert to currency then round again
    Result := ConvertTo (Result, ATo);
    Result := EuroRound (Result);
  end;
end;
```

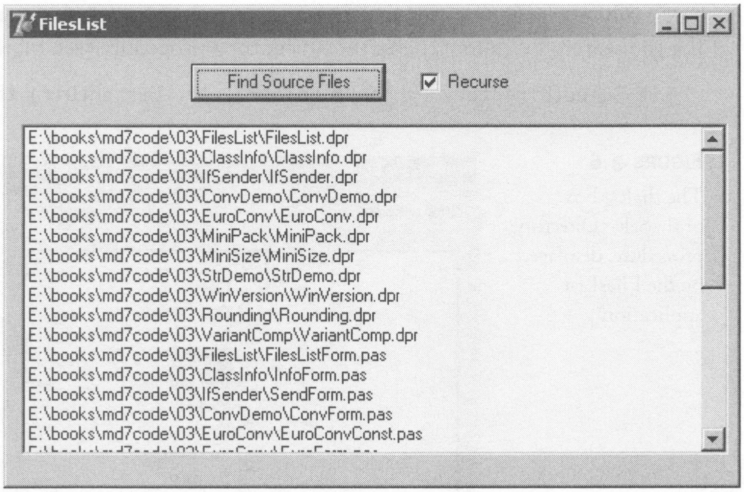
Of course, you might want to extend the example by providing conversion to other non-euro currencies, eventually picking the values automatically from a website. I'll leave this as a rather complex exercise for you.

Managing Files with SysUtils

To access files and file information, you can generally rely on the standard functions available in the SysUtils unit. Relying on these fairly traditional Pascal libraries makes your code easily portable among quite different operating systems (although you'll have to consider with great care the differences in the file system architectures, particularly case sensitivity on the Linux platform).

For example, the FilesList example uses the `FindFirst`, `FindNext`, and `FindClose` combination to retrieve from within a folder a list of files that match a filter, with the same code you could use on Kylix and Linux (an example of the output appears in Figure 3.5).

FIGURE 3.5
An example of the output of the FilesList application



```
E:\books\md7code\03\FilesList\FilesList.dpr
E:\books\md7code\03\ClassInfo\ClassInfo.dpr
E:\books\md7code\03\IfSender\IfSender.dpr
E:\books\md7code\03\ConvDemo\ConvDemo.dpr
E:\books\md7code\03\EuroConv\EuroConv.dpr
E:\books\md7code\03\MiniPack\MiniPack.dpr
E:\books\md7code\03\MiniSize\MiniSize.dpr
E:\books\md7code\03\StrDemo\StrDemo.dpr
E:\books\md7code\03\WinVersion\WinVersion.dpr
E:\books\md7code\03\Rounding\Rounding.dpr
E:\books\md7code\03\VariantComp\VariantComp.dpr
E:\books\md7code\03\FilesList\FilesListForm.pas
E:\books\md7code\03\ClassInfo\InfoForm.pas
E:\books\md7code\03\IfSender\SendForm.pas
E:\books\md7code\03\ConvDemo\ConvForm.pas
E:\books\md7code\03\EuroConv\EuroConvConst.pas
```

The following code adds the filenames to a list box called `lbFiles`:

```
procedure TForm1.AddFilesToList(Filter, Folder: string;
  Recurse: Boolean);
var
  sr: TSearchRec;
begin
  if FindFirst (Folder + Filter, faAnyFile, sr) = 0 then
  repeat
    lbFiles.Items.Add (Folder + sr.Name);
  until FindNext(sr) <> 0;
  FindClose(sr);
```

If the `Recurse` parameter is set, the `AddFilesToList` procedure gets a list of subfolders by examining the local files again, and then calls itself for each of the subfolders. The list of folders is placed in a string list object, with the following code:

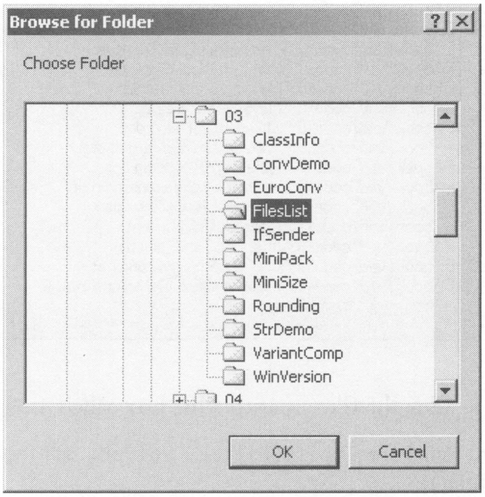
```
procedure GetSubDirs (Folder: string; sList: TStringList);
var
  sr: TSearchRec;
begin
  if FindFirst (Folder + '*.*', faDirectory, sr) = 0 then
  try
    repeat
      if (sr.Attr and faDirectory) = faDirectory then
        sList.Add (sr.Name);
    until FindNext(sr) <> 0;
  finally
    FindClose(sr);
  end;
end;
```

Finally, the program uses an interesting technique to ask the user to select the initial directory for the file search, by calling the `SelectDirectory` procedure (see Figure 3.6):

```
if SelectDirectory ('Choose Folder', '', CurrentDir) then ...
```

FIGURE 3.6

The dialog box of the SelectDirectory procedure, displayed by the FilesList application



The *TObject* Class

As mentioned earlier, a key element of the System unit is the definition of the `TObject` class, which is the *mother of all Delphi classes*. Every class in the system inherits from the `TObject` class, either directly (if you specify `TObject` as the base class), implicitly (if you indicate no base class), or indirectly (when you specify another class as the ancestor). The entire hierarchy of classes in an Object Pascal program has a single root. So, you can use the `TObject` data type as a replacement for the data type of any class type in the system, according to the type compatibility rules covered in Chapter 2 in the section "Inheritance and Type Compatibility."

For example, components' event handlers usually have a `Sender` parameter of type `TObject`. This simply means that the `Sender` object can be of any class, because every class is ultimately derived from `TObject`. The typical drawback of such an approach is that to work on the object, you need to know its data type. In fact, when you have a variable or a parameter of the `TObject` type, you can apply to it only the methods and properties defined by the `TObject` class itself. If this variable or parameter happens to refer to an object of the `TButton` type, for example, you cannot directly access its `Caption` property. The solution to this problem lies in the use of the safe down-casting or run-time type information (RTTI) operators (`is` and `as`) discussed in Chapter 2.

You can also use another approach. For any object, you can call the methods defined in the `TObject` class. For example, the `ClassName` method returns a string with the name of the class. Because it is a class method (see Chapter 2 for details), you can apply it both to an object and to a class. Suppose you have defined a `TButton` class and a `Button1` object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

On some occasions you need to use the name of a class, but it can also be useful to retrieve a class reference to the class itself or to its base class. The class reference allows you to operate on the class at run time (as you saw in the preceding chapter), whereas the class name is just a string. You can get these class references with the `ClassType` and `ClassParent` methods. The first returns a class reference to the class of the object; the second returns a class reference to the object's base class. Once you have a class reference, you can apply to it any class methods of `TObject`—for example, to call the `ClassName` method.

Another method that might be useful is `InstanceSize`, which returns the run-time size of an object. Although you might think that the `SizeOf` global function provides this information, that function actually returns the size of an object reference—a pointer, which is invariably four bytes—instead of the size of the object itself.

In Listing 3.1, you can find the complete definition of the `TObject` class, extracted from the System unit. In addition to the methods I've already mentioned, notice `InheritsFrom`, which provides a test that's similar to the `is` operator but that can also be applied to classes and class references (the first argument of `is` must be an object).

LISTING 3.1: THE DEFINITION OF THE *TOBJECT* CLASS (IN THE SYSTEM RTL UNIT)

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(
      const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID;out Obj): Boolean;
    class function GetInterfaceEntry(
      const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

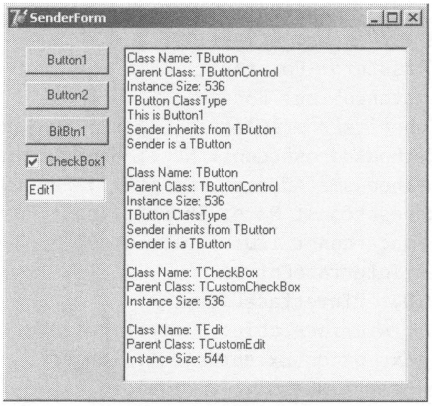
NOTE *The* `ClassInfo` *method returns a pointer to the internal run-time type information (RTTI) of the class, introduced in the next chapter.*

These methods of `TObject` are available for objects of every class, because `TObject` is the common ancestor class of every class. Here is how you can use these methods to access class information:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
  Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);

  if Sender.ClassParent <> nil then
    Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);

  Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
end;
```

The code checks to see whether the `ClassParent` is `nil`, in case you are using an instance of the `TObject` type, which has no base type.

This `ShowSender` method is part of the IfSender example. The method is connected with the `OnClick` event of several controls: three buttons, a check box, and an edit box. When you click each control, the `ShowSender` method is invoked with the corresponding control as sender (more on events in Chapter 4). One of the buttons is a Bitmap button, an object of a `TButton` subclass. You can see an example of the output of this program at run time in Figure 3.7.



```
Class Name: TButton
Parent Class: TButtonControl
Instance Size: 536
TButton ClassType
This is Button1
Sender inherits from TButton
Sender is a TButton

Class Name: TButton
Parent Class: TButtonControl
Instance Size: 536
TButton ClassType
Sender inherits from TButton
Sender is a TButton

Class Name: TCheckBox
Parent Class: TCustomCheckBox
Instance Size: 536

Class Name: TEdit
Parent Class: TCustomEdit
Instance Size: 544
```

FIGURE 3.7
The output of the IfSender example

You can use other methods to perform tests. For example, you can check whether the `Sender` object is of a specific type with the following code:

```
if Sender.ClassType = TButton then ...
```

You can also check whether the `Sender` parameter corresponds to a given object, with this test:

```
if Sender = Button1 then...
```

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class *or* one of its subclasses. Doing so lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the `InheritsFrom` method, which is also called when you use the `is` operator. The following two tests are equivalent:

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

Showing Class Information

I've extended the IfSender example to show a complete list of base classes of a given object or class. Once you have a class reference you can add all of its base classes to the `ListParent` list box with the following code:

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
```

You'll notice that I use a class reference at the heart of the `while` loop, which tests for the absence of a parent class (so that the current class is `TObject`). Alternatively, I could have written the `while` statement in either of the following ways:

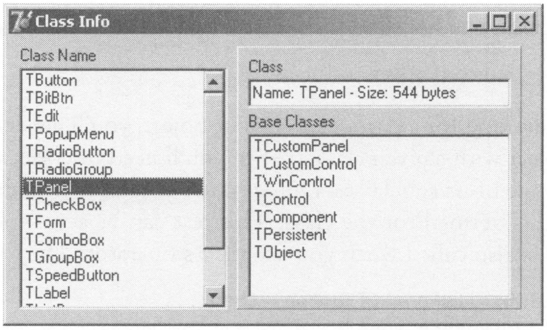
```
while not MyClass.ClassNameIs ('TObject') do...
while MyClass <> TObject do...
```

The code in the `with` statement referring to the `ListParent` list box is part of the ClassInfo example, which displays the list of parent classes and some other information about a few components of the VCL (basically those on the Standard page of the Component Palette). These components are manually added to a dynamic array holding classes and declared as

```
private
  ClassArray: array of TClass;
```

When the program starts, the array is used to show all the class names in a list box. Selecting an item from the list box triggers the visual presentation of its details and its base classes, as you can see in the program output in Figure 3.8.

FIGURE 3.8
The output of the ClassInfo example


Class Info
Class Name
TButton
TBitBtn
TEdit
TPopupMenu
TRadioButton
TRadioGroup
TPanel
TCheckBox
TForm
TComboBox
TGroupBox
TSpeedButton
TLabel
Class
Base Classes
TCustomPanel
TCustomControl
TWinControl
TControl
TComponent
TPersistent
TObject

Name: TPanel - Size: 544 bytes

NOTE *As a further extension of this example, you can create a tree with all the base classes of the various components in a hierarchy. To do that, I've created the VclHierarchy wizard, discussed in Appendix A ("Extra Delphi Tools by the Author").*

What's Next?

In this chapter, I've focused my attention on new features of the Delphi function-based run-time library. I have provided only a summary of the RTL, not a complete overview (which would take too much space). You can find more examples of the basic RTL functions of Delphi in the free e-books available on my website, as discussed in Appendix C.

In the next chapter, we'll begin moving from the function-based RTL to the class-based RTL, which is the core of Delphi's class library. I won't debate whether the core classes common to the VCL and CLX, such as `TObject`, actually belong to the RTL or the class library. I've covered everything defined in System, SysUtils, and other units hosting functions and procedures in this chapter; the next chapter focuses on the Classes unit and other core units that define classes.

Along with the preceding chapter on the Delphi language, Chapter 4 will provide a foundation for discussing visual- and database-oriented classes (or components, if you prefer). Looking at the various library units, you'll find many more global functions, which don't belong to the core RTL but are still quite useful.

Chapter 4

Core Library Classes

WE SAW IN THE preceding chapter that Delphi includes a large number of functions and procedures, but the real power of Delphi's visual programming lies in the huge class library that comes with it. Delphi's standard class library contains hundreds of classes, with thousands of methods, and it is so large that I certainly cannot provide a detailed reference in this book. What I'll do, instead, is explore various areas of this library starting with this chapter and continuing through those that follow.

This chapter is devoted to the library's core classes as well as to some standard programming techniques, such as the definition of events. We'll explore some commonly used classes, such as lists, string lists, collections, and streams. We'll devote most of our time to exploring the content of the Classes unit, but we'll also examine other core units of the library.

Delphi classes can be used either entirely in code or within the visual form designer. Some of them are component classes, which show up in the Component Palette, and others are more general-purpose. The terms *class* and *component* can be used almost as synonyms in Delphi. Components are the central elements of Delphi applications. When you write a program, you basically choose a number of components and define their interactions—that's all there is to Delphi visual programming.

Before reading this chapter, you need to have a good understanding of the language, including inheritance, properties, virtual methods, class references, and so on, as discussed in Chapter 2, "The Delphi Programming Language."

This chapter covers the following topics:

- The RTL package, CLX, and VCL
- *TPersistent* and *published*
- The *TComponent* base class and its properties
- Components and ownership
- Events
- Lists, container classes, and collections
- Streaming
- The units of the RTL package

The RTL Package, VCL, and CLX

Until version 5, Delphi's class library was known as VCL, which stands for Visual Components Library. This is a component library mapped on top of the Windows API. Kylix, the Delphi version for Linux, introduced a new component library, called CLX (Component Library for X-Platform or Cross Platform; the acronym is pronounced "clicks"). Delphi 6 was the first edition to include both the VCL and CLX libraries. For visual components, the two class libraries are alternative one to the other. However, the core classes and the database and Internet portions of the two libraries are basically shared.

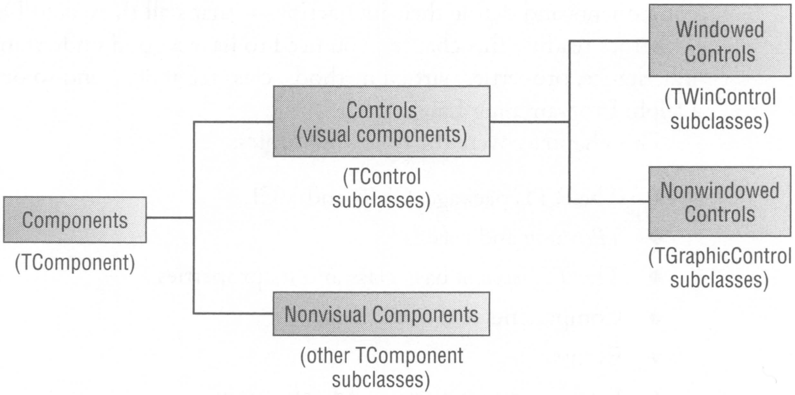
VCL was considered as a single large library, although programmers used to refer to different parts of it (components, controls, nonvisual components, data sets, data-aware controls, Internet components, and so on). CLX introduces a distinction in four parts: BaseCLX, VisualCLX, DataCLX, and NetCLX. Only in VisualCLX does the library use a totally different approach between the Windows and Linux platforms; the rest of the code is inherently portable to Linux. In the following section, I discuss these two libraries; the rest of the chapter focuses on the common core classes.

In recent versions of Delphi, this distinction is underlined by the fact that the core non-visual components and classes of the library are part of the new RTL package, which is used by both VCL and CLX. Moreover, using this package in non-visual applications (for example, web server programs) allows you to reduce considerably the size of the files to deploy and load in memory.

Traditional Sections of VCL

Delphi programmers used to refer to the sections of the VCL with names Borland originally suggested in its documentation—names that became common afterward for different groups of components. Technically, components are subclasses of the `TComponent` class, which is one of the root classes of the hierarchy, as you can see in Figure 4.1. The `TComponent` class inherits from the `TPersistent` class; the role of these two classes will be explained in the next section.

FIGURE 4.1
A graphical representation of the main groups of VCL components



In addition to components, the library includes classes that inherit directly from `TObject` and from `TPersistent`. These classes are collectively known as *Objects* in portions of the documentation, a rather confusing name. These noncomponent classes are often used for values of properties or as utility classes used in code; not inheriting from `TComponent`, these classes cannot be used directly in visual programming.

NOTE *To be more precise, noncomponent classes cannot be made available in the Component Palette and cannot be dropped directly into a form, but they can be visually managed with the Object Inspector as subproperties of other properties or items of collections of various types. So, even noncomponent classes are often easily used when interacting with the Form Designer.*

The component classes can be further divided into two main groups: controls and nonvisual components.

Controls All the classes that descend from `TControl`. Controls have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different subspecifications—window-based or graphical—that I'll discuss in more detail in Chapter 5, "Visual Controls."

Nonvisual Components All the components that are not controls—all the classes that descend from `TComponent` but not from `TControl`. At design time, a nonvisual component appears on the form or data module as an icon, with a caption below it (the caption is optional on forms). At run time, some of these components may be visible (for example, the standard dialog boxes), and others are always invisible (for example, the database table component).

TIP *You can simply move the mouse cursor over a control or component in the Form Designer to see a Tooltip with its name and class type (and some extended information). You can also use an environment option, Show Component Captions, to see the name of a nonvisual component under its icon.*

The traditional subdivision of VCL is very common for Delphi programmers. Even with the introduction of CLX and some new naming schemes, the traditional names will probably survive and merge into Delphi programmers' jargon.

The Structure of CLX

Borland now refers to different portions of the CLX library using one terminology under Linux and a slightly different (and less clear) naming structure in Delphi. This new subdivision of the cross-platform library represents more logical areas than the structure of the class hierarchy:

BaseCLX The core of the class library: the topmost classes (such as `TComponent`) and several general utility classes (including lists, containers, collections, and streams). Compared to the corresponding classes of VCL, BaseCLX is largely unchanged and is highly portable between the Windows and Linux platforms. This chapter is largely devoted to exploring BaseCLX and the common VCL core classes.

VisualCLX The collection of visual components, generally called controls. This is the portion of the library that is more tightly related to the operating system: VisualCLX is implemented on top of the Qt library, available both on Windows and on Linux. Using VisualCLX allows for full portability of the visual portion of your application between Delphi on Windows and Kylix on Linux. However, most of the VisualCLX components have corresponding VCL controls, so you can also easily move your code from one library to the other. I'll discuss VisualCLX and the controls of VCL in Chapter 5.

DataCLX All the database-related components of the library. DataCLX is the front end of the new dbExpress database engine included in both Delphi and Kylix. Delphi also includes the traditional BDE front end, dbGo, and InterBase Express (IBX). If we consider all these components as part of DataCLX, only the dbExpress front end and IBX are portable between Windows and Linux. In addition, DataCLX includes the ClientDataSet component, now called MyBase, and other related classes. Delphi's data access components are discussed in Part III of the book.

NetCLX The Internet-related components, from the WebBroker framework to the HTML producer components, from Indy (Internet Direct) to Internet Express, from WebSnap to XML support. This part of the library is, again, highly portable between Windows and Linux. Internet support is discussed in Part IV of the book. (The name is short for Internet CLX, and has nothing to do with the Microsoft .NET technology it predates.)

VCL-Specific Sections of the Library

The preceding areas of the library are available, with the differences I've mentioned, on both Delphi and Kylix. In Delphi, however, other sections of the VCL are for one reason or another specific to Windows only:

- The Delphi ActiveX (DAX) framework provides support for COM, OLE Automation, ActiveX, and other COM-related technologies. See Chapter 12, "From COM to COM+," for more information on this area of Delphi.
- The Decision Cube components provide Online Analytical Processing (OLAP) support but have ties with the BDE and haven't been updated recently. Decision Cube is not discussed in the book.

Finally, the default Delphi installation includes some third-party components, such as TeeChart for business graphics, RAVE for reporting and printing, and IntraWeb for Internet development. Some of these components will be discussed in the book, but they are not strictly part of the VCL. RAVE and IntraWeb are also available for Kylix.

The *TPersistent* Class

The first core class of the Delphi library we'll look at is `TPersistent`, which is quite a strange class: It has very little code and almost no direct use, but it provides a foundation for the entire idea of visual programming. You can see the definition of the class in Listing 4.1.

LISTING 4.1: THE DEFINITION OF THE *TPERSISTENT* CLASS, FROM THE CLASSES UNIT

```
{$M+}
TPersistent = class(TObject)
private
  procedure AssignError(Source: TPersistent);
protected
  procedure AssignTo(Dest: TPersistent); virtual;
  procedure DefineProperties(Filer: TFiler); virtual;
  function  GetOwner: TPersistent; dynamic;
public
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); virtual;
  function  GetNamePath: string; dynamic;
end;
```

As the name implies, this class handles persistency—that is, saving the value of an object to a file to be used later to re-create the object in the same state and with the same data. Persistency is a key element of visual programming. In fact (as you saw in Chapter 1, "Delphi 7 and Its IDE"), at design time in Delphi you manipulate actual objects, which are saved to DFM files and re-created at run time when the specific component container—form or data module—is created.

NOTE *Everything I say about DFM files also applies to XFM files, the file format used by CLX applications. The format is identical. The extension difference is relevant because Delphi uses it to determine whether the form is based on CLX/Qt or on VCL/Windows. In Kylix, every form is a CLX/Qt form, regardless of which extension is used; so, the XFM/DFM file extension in Kylix really doesn't matter.*

Streaming support is not embedded in the `TPersistent` class but is provided by other classes, which target `TPersistent` and its descendants. In other words, you can "persist" with Delphi default streaming-only objects of classes inheriting from `TPersistent`. One of the reasons for this behavior lies in the fact that the class is compiled with a special option turned on, `{$M+}`. This flag activates the generation of extended RTTI information for the published portion of the class.

Delphi's streaming system doesn't try to save the in-memory data of an object, which would be complex because of the many pointers to other memory locations and totally meaningless when the object was reloaded. Instead, Delphi saves objects by listing the values of all properties in the `published` section of the class. When a property refers to another object, Delphi saves the name of the object or the entire object (with the same mechanism) depending on its type and relationship with the main object. For a comparison with other approaches, see the sidebar "Object Streaming versus Code Generation."

The only method of the `TPersistent` class that you'll generally use is the `Assign` procedure, which can be used to copy the actual value of an object. In the library, this method is implemented by many noncomponent classes but by very few components. Most subclasses reimplement the virtual protected `AssignTo` method, called by the default implementation of `Assign`.

Other methods include `DefineProperties`, used for customizing the streaming system and adding extra information (pseudo-properties); and the `GetOwner` and `GetNamePath` methods, used by collections and other special classes to identify themselves to the Object Inspector.

OBJECT STREAMING VERSUS CODE GENERATION

The approach used by Delphi (and Kylix) is different from the approach used by other visual development tools and languages. For example, in Java, the effect of the definition of a form inside an IDE is the generation of the Java source code used to create the components and set their properties. Setting properties in an inspector affects the source code. Something similar happens in C#, although properties in this language are closer to the notion of properties in Delphi. You've already seen that in Delphi; you can write code to generate the components instead of relying on streaming, but because there is no specific support in the IDE you'll have to write that code manually.

Continued on next page

OBJECT STREAMING VERSUS CODE GENERATION *(continued)*

Each of the two approaches has advantages and disadvantages. When generating source code, you have more control over what goes on and the exact sequence of creation and initialization. Delphi reloads objects and their properties but delays some assignments until a later fix-up phase, to avoid the problems of references to not-yet-initialized objects. This process is more complex, but it is so hidden that it becomes simpler for the programmer.

The Java language allows a tool like JBuilder to recompile a form class and load it in a running program for every change. In a compiled system like Delphi, this approach would be more complex (Delphi uses a fake version, technically called *a proxy*, of your form at design time, not the actual form).

One advantage of the approach used by Delphi is that the DFM files can be translated into different languages without affecting the source code; this is one reason Java is offering XML persistence of forms. Another difference is that Delphi embeds the component's graphic in the DFM file, instead of referring to external files. Doing so simplifies deployment (because everything ends up in the executable file) but can also make the executable much bigger.

The *published* Keyword

Delphi has four directives specifying data access: `public`, `protected`, `private`, and `published`. I've covered the first three in Chapter 2, "The Delphi Programming Language," so now it's time to look at what `published` means. For any published field, property, or method, the compiler generates extended RTTI information, so that Delphi's run-time environment or a program can query a class for its published interface. For example, every Delphi component has a published interface that is used by the IDE, in particular the Object Inspector. A proper use of published items is important when you write components. Usually, the published part of a component contains no fields or methods, just properties and events.

When Delphi generates a form or data module, it places the definitions of its components and methods (the event handlers) in the first portion of its definition, before the `public` and `private` keywords. These fields and methods in the initial portion of the class are published. The default is `published` when no special keyword is added before an element of a component class.

To be more precise, `published` is the default keyword only if the class was compiled with the `$M+` compiler directive or is descended from a class compiled with `$M+`. This directive is used in the `TPersistent` class, so most classes of the VCL and all the component classes default to `published`. However, noncomponent classes in Delphi (such as `TStream` and `TList`) are compiled with `$M-` and default to public visibility.

The methods used to handle events in the IDE (and in DFM files) should be published, and the fields corresponding to your components in the form should be published to be automatically connected with the objects described in the DFM file and created along with the form. (Later in this chapter I'll discuss the details of this situation and the problems it generates.)

ACCESSING PUBLISHED FIELDS AND METHODS

As I've mentioned, three different declarations make sense in the `published` section of a class: fields, methods, and properties. In your code, you'll generally refer to published items like you refer to public ones, that is, by referring to the corresponding identifiers in the code. In some special cases, though, it is possible to access published items at runtime by name. I'll discuss dynamic access to properties in the section "Accessing Properties by Name;" here I'll introduce possible ways of interacting at runtime with fields and methods. The `TObject` class has three interesting methods for this area: `MethodAddress`, `MethodName`, and `FieldAddress`.

The first function, `MethodAddress`, returns the memory address of the compiled code (a sort of function pointer) of the method passed as parameter in a string. By assigning this method address to the Code field of a `TMethod` structure and assigning an object to the `Data` field, you can obtain a complete method pointer. At this point, to call the method you must cast it to the proper method pointer type. Here is a code fragment highlighting the key points of this technique:

```
var
  Method: TMethod;
  Evt: TNotifyEvent;
begin
  Method.Code := MethodAddress ('Button1Click');
  Method.Data := Self;
  Evt := TNotifyEvent(Method);
  Evt (Sender); // call the method
end;
```

Delphi uses similar code to assign an event handler when it loads a DFM file, because these files store the name of the methods used to handle the events, whereas the components store the method pointer. The second method, `MethodName`, does the opposite transformation, returning the name of the method at a given memory address. This method can be used to obtain the name of an event handler, given its value, something Delphi does when streaming a component into a DFM file.

Finally, the `FieldAddress` method of `TObject` returns the memory location of a published field, given its name. Delphi uses this method to connect components created from the DFM files with the fields of their owner (for example, a form) having the same name.

Note that these three methods are seldom used in "normal" programs but play a key role in making Delphi work. They are strictly related to the streaming system. You'll need to use these methods only when writing extremely dynamic programs, special-purpose wizards, or other Delphi extensions.

Accessing Properties by Name

The Object Inspector displays a list of an object's published properties, even for components you've written. To do this, it relies on the RTTI information generated for published properties. Using some advanced techniques, an application can retrieve a list of an object's published properties and use them.

Although this capability is not very well known, in Delphi it is possible to access properties by name simply by using the string with the name of the property and then retrieving its value. Access to the RTTI information of properties is provided through a group of undocumented subroutines, part of the TypInfo unit.

WARNING *These subroutines have always been undocumented in past versions of Delphi, so that Borland remained free to change them. However, from Delphi 1 to Delphi 7, changes were very limited and related only to supporting new features, with a high level of backward compatibility. In Delphi 5, Borland added many more goodies and a few "helper" routines that are officially promoted (even if still not fully documented in the Help file but explained only with comments provided in the unit).*

Rather than explore the entire TypInfo unit here, we will look at only the minimal code required to access properties by name. Prior to Delphi 5, it was necessary to use the `GetPropInfo` function to retrieve a pointer to some internal property information and then apply one of the access functions, such as `GetStrProp`, to this pointer. You also had to check for the existence and the type of the property.

Now you can use a new set of TypInfo routines, including the handy `GetPropValue`, which returns a variant with the value of the property and raises an exception if the property doesn't exist. To avoid the exception, you can call the `IsPublishedProp` function first. You simply pass to these functions the object and a string with the property name. A further optional parameter of `GetPropValue` allows you to choose the format for returning values of properties of any set type (either a string or the numeric value for the set). For example, you can call

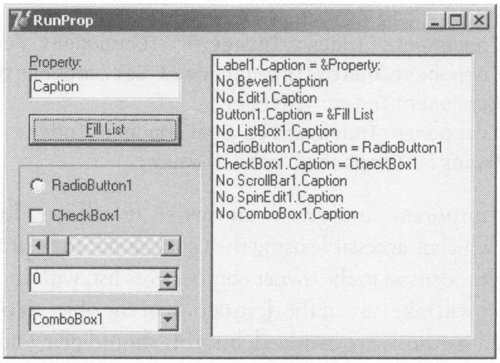
```
ShowMessage (GetPropValue (Button1, 'Caption'));
```

This call has the same effect as calling `ShowMessage`, passing as parameter `Button1.Caption`. The only real difference is that this version of the code is much slower, because the compiler generally resolves normal access to properties in a more efficient way. The advantage of the run-time access is that you can make it very flexible, as in the following RunProp example.

This program displays in a list box the value of a property of any type for each component of a form. The name of the property you are looking for is provided in an edit box. Being able to type the property name in the edit box makes the program very flexible. In addition to the edit box and the list box, the form has a button to generate the output and some other components added only to test their properties. When you click the button, the following code is executed:

```
uses
  TypInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Value: Variant;
begin
  ListBox1.Clear;
  for I := 0 to ComponentCount -1 do
  begin
    if IsPublishedProp (Components[I], Edit1.Text) then
    begin
      Value := GetPropValue (Components[I], Edit1.Text);
      ListBox1.Items.Add (Components[I].Name + '.' +
        Edit1.Text + ' = ' + string (Value));
    end
    else
     ListBox1.Items.Add ('No ' + Components[I].Name + '.' +
      Edit1.Text);
   end;
  end;
```

Figure 4.2 shows the effect of clicking the Fill List button while using the default `Caption` value in the edit box. You can try it with any other property name. Numbers will be converted to strings by the variant conversion. Objects (such as the value of the `Font` property) will be displayed as memory addresses.



Property: Caption

```
Label1.Caption = &Property:
No Bevel1.Caption
No Edit1.Caption
Button1.Caption = &Fill List
No ListBox1.Caption
RadioButton1.Caption = RadioButton1
CheckBox1.Caption = CheckBox1
No ScrollBar1.Caption
No SpinEdit1.Caption
No ComboBox1.Caption
```

FIGURE 4.2
The output of the RunProp example, which accesses properties by name at run time

WARNING *Do not use regularly the TypInfo unit instead of polymorphism and other property-access techniques. Use base-class property access first, or use the safe* `as` *typecast when required, and reserve RTTI access to properties as a last resort. Using TypInfo techniques makes your code slower, more complex, and more prone to human error; in fact, it skips the compile-time type-checking.*

The *TComponent* Class

If the `TPersistent` class is more important than it seems at first sight, the key class at the heart of Delphi's component-based class library is `TComponent`, which inherits from `TPersistent` (and from `TObject`). The `TComponent` class defines many core elements of components; however, it is not as complex as you might think, because the base classes and the language already provide most of what's needed.

I won't explore all the details of the `TComponent` class, some of which are more important for component designers than they are for component users. I'll just discuss ownership (which accounts for some public properties of the class) and the two published properties of the class, `Name` and `Tag`.

Ownership

One of the core features of the `TComponent` class is the definition of ownership. When a component is created, it can be assigned an owner component, which will be responsible for destroying it. So, every component can have an owner and can also be the owner of other components.

Several public methods and properties of the class are devoted to handling the *two sides* of ownership. Here is a list, extracted from the class declaration (in the Classes unit of the VCL):

```
type
  TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
  public
    constructor Create(AOwner: TComponent); virtual;
    procedure DestroyComponents;
    function FindComponent(const AName: string): TComponent;
    procedure InsertComponent(AComponent: TComponent);
    procedure RemoveComponent(AComponent: TComponent);

    property Components[Index: Integer]: TComponent read GetComponent;
    property ComponentCount: Integer read GetComponentCount;
    property ComponentIndex: Integer
      read GetComponentIndex write SetComponentIndex;
    property Owner: TComponent read FOwner;
```

If you create a component and give it an owner, it will be added to the list of components (`InsertComponent`), which is accessible using the `Components` array property. The specific component has an `Owner` and knows its position in the owner components list, with the `ComponentIndex` property. Finally, the owner's destructor will take care of the destruction of the object it owns by calling `DestroyComponents`. A few more protected methods are involved, but this should give you the overall picture.

It's important to emphasize that component ownership can solve many of your applications' memory management problems, if used properly. When you use the Form Designer or Data Module Designer of the IDE, that form or data module will own any component dropped on it. At the same time, you should generally create components with a form or data module owner, even in your code. In these circumstances you only need to remember to destroy the component containers (form or data module) when they are not needed anymore, and you can forget about the components they contain. For example, you delete a form to destroy all the components it contains at once, which is a major simplification compared to having to remember to free each and every object individually. At a larger scale, forms and data modules are generally owned by the `Application` object, which is destroyed by the VCL shutdown code freeing all of the component containers, which free the components they contain.

THE COMPONENTS ARRAY

The `Components` property can also be used to access one component owned by another—let's say, a form. This property can be very handy (compared to using a specific component directly) for writing generic code, acting on all or many components at a time. For example, you can use the following code to add to a list box the names of all a form's components (this code is part of the ChangeOwner example presented in the next section):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
```

```
    for I := 0 to ComponentCount - 1 do
      ListBox1.Items.Add (Components [I].Name);
  end;
```

This code uses the `ComponentCount` property, which holds the total number of components owned by the current form, and the `Components` property, which is the list of owned components. When you access a value from this list, you get a value of the `TComponent` type. For this reason, you can directly use only the properties common to all components, such as the `Name` property. To use properties specific to particular components, you have to use the proper type-downcast (`as`).

NOTE *In Delphi, some components are also component containers: the GroupBox, Panel, PageControl, and, of course, Form components. When you use these controls, you can add other components inside them. In this case, the container is the parent of the components (as indicated by the* `Parent` *property), and the form is their owner (as indicated by the* `Owner` *property). You can use the* `Controls` *property of a form or group box to navigate the child controls, and you can use the* `Components` *property of the form to navigate all the owned components, regardless of their parent.*

Using the `Components` property, you can always access each component of a form. If you need access to a specific component, however, instead of comparing each name with the name of the component you are looking for, you can let Delphi do this work by using the form's `FindComponent` method. This method simply scans the `Components` array looking for a name match. More information about the role of the `Name` property for a component is in the section "The *Name* Property."

CHANGING THE OWNER

You have seen that almost every component has an owner. When a component is created at design time (or from the resulting DFM file), its owner will invariably be its form. When you create a component at run time, the owner is passed as a parameter to the `Create` constructor.

`Owner` is a read-only property, so you cannot change it. The owner is set at creation time and should generally not change during the lifetime of a component. To understand why you should not change a component's owner at design time nor freely change its name, read the following discussion. Be warned that the topic covered is not simple; if you're just beginning with Delphi, you might want to come back to this section at a later time.

To change the owner of a component, you can call the `InsertComponent` and `RemoveComponent` methods of the owner itself, passing the current component as parameter. However, you cannot apply these methods directly in a form's event handler, as I attempt to do here:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveComponent (Button1);
  Form2.InsertComponent (Button1);
end;
```

This code produces a memory access violation, because when you call `RemoveComponent`, Delphi disconnects the component from the form field (`Button1`), setting it to `nil`. (I talk more about form fields in the section "Removing Form Fields.") The solution is to write a procedure like this:

```
procedure ChangeOwner (Component, NewOwner: TComponent);
begin
```

```
    Component.Owner.RemoveComponent (Component);
    NewOwner.InsertComponent (Component);
  end;
```

This method (extracted from the ChangeOwner example) changes the owner of the component. It is called along with the simpler code used to change the parent component; the two commands combined move the button *completely* to another form, changing its owner:

```
procedure TForm1.ButtonChangeClick(Sender: TObject);
begin
  if Assigned (Button1) then
  begin
    // change parent
    Button1.Parent := Form2;
    // change owner
    ChangeOwner (Button1, Form2);
  end;
end;
```

The method checks whether the `Button1` field still refers to the control, because while moving the component, Delphi will set `Button1` to `nil`. You can see the effect of this code in Figure 4.3.

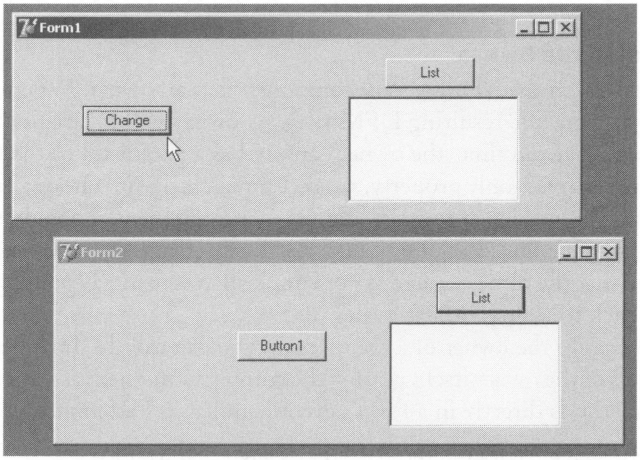


FIGURE 4.3
In the ChangeOwner example, clicking the Change button moves the `Button1` component to the second form.

To demonstrate that the owner of the `Button1` component actually changes, I've added another feature to both forms. The List button fills the list box with the names of the components each form owns, using the procedure shown in the previous section. Click the two List buttons before and after moving the component, and you'll see what happens behind the scenes. As a final feature, the `Button1` component has a simple handler for its `OnClick` event, to display the caption of the owner form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage ('My owner is ' + ((Sender as TButton).Owner as TForm).Caption);
end;
```

The *Name* Property

Every component in Delphi should have a name. The name must be unique within the owner component, which is generally the form into which you place the component. This means an application can have two different forms, each with a component that has the same name, although you might want to avoid this practice to prevent confusion. It is generally better to keep component names unique throughout an application.

Setting a proper value for the `Name` property is very important: If it's too long, you'll need to type a lot of code to use the object; if it's too short, you may confuse different objects. Usually the name of a component has a prefix with the component type; this makes the code more readable and allows Delphi to group components in the combo box of the Object Inspector, where they are sorted by name. Three important elements are related to a component's `Name` property:

- At design time, the value of the `Name` property is used to define the name of the form field in the declaration of the form class. This is the name you'll generally use in the code to refer to the object. For this reason, the value of the `Name` property must be a legal Delphi language identifier (it must have no spaces and begin with a letter, not a number).
- If you set a control's `Name` property before changing its `Caption` or `Text` property, the new name is often copied to the caption. That is, if the name and the caption are identical, then changing the name will also change the caption.
- Delphi uses the name of the component to create the default name of the methods related to its events. If you have a `Button1` component, its default `OnClick` event handler will be called `Button1Click` unless you specify a different name. If you later change the name of the component, Delphi will modify the names of the related methods accordingly. For example, if you change the name of the button to `MyButton`, the `Button1Click` method automatically becomes `MyButtonClick`.

As mentioned earlier, if you have a string with the name of a component, you can get its instance by calling the `FindComponent` method of its owner, which returns `nil` if the component is not found. For example, you can write

```
var
  Comp: TComponent;
begin
  Comp := FindComponent ('Button1');
  if Assigned (Comp) then
    with Comp as TButton do
      // some code...
```

NOTE *Delphi also includes a* `FindGlobalComponent` *function, which finds a top-level component (a form or data module) that has a given name.* `FindGlobalComponent` *calls one or more installed functions, so in theory you can modify the way the function works. However, because* `FindGlobalComponent` *is used by the streaming system, I strongly recommend against installing your own replacement functions. If you want a customized way to search for components on other containers, simply write a new function with a custom name.*

Removing Form Fields

Every time you add a component to a form, Delphi adds an entry for it, along with some of its properties, to the DFM file. To the Pascal file, Delphi adds the corresponding field in the form class declaration. This field of the form is a reference to the corresponding object, as is any class-type variable in Delphi. When the form is created, Delphi loads the DFM file and uses it to re-create all the components and set their properties back to the design-time values, saved in the DFM file itself. Then it connects the new object with the form field corresponding to its `Name` property. This is why in your code, you can use the form field to operate on the corresponding component.

For this reason, it is possible to have a component without a name. If your application will not manipulate the component or modify it at run time, you can remove the component name from the Object Inspector. Examples include a static label with fixed text, or a menu item, or even more obviously, menu item separators. By blanking out the name, you remove the corresponding element from the form class declaration. Doing so reduces the size of the form object (by only four bytes, the size of the object reference) and reduces the DFM file by not including a useless string (the component name). Reducing the DFM file size also implies reducing the final executable file size, even if only slightly.

WARNING *If you remove component names, just make sure to leave at least one named component of each class used on the form, so the smart linker and the streaming system will link in the required code for the class and recognize it from the DFM file. For example, if you remove from a form all the fields referring to* `TLabel` *components, when the system loads the form at run time, it will be unable to create an object of an unknown class and will issue an error indicating that the class is not available. As we'll see in the next section you can call the* `RegisterClass` *or* `RegisterClasses` *routines to avoid such an error.*

You can also keep the component name and manually remove the corresponding field of the form class. Even if the component has no corresponding form field, it is created anyway, although using it (through the `FindComponent` method, for example) will be a little more difficult.

Hiding Form Fields

Many OOP purists complain that Delphi doesn't really follow the encapsulation rules, because all the components of a form are mapped to public fields and can be accessed from other forms and units. Fields for components are listed in the first unnamed section of a class declaration, which has a default visibility of published. However, Delphi does that only as a default to help beginners learn to use the Delphi visual development environment quickly. A programmer can follow a different approach and use properties and methods to operate on forms. The risk, however, is that another programmer on the same team might inadvertently bypass this approach, directly accessing the components if they are left in the published section. The solution, which many programmers don't know about, is to move the components to the private portion of the class declaration.

As an example, I've made a simple form with an edit box, a button, and a list box. When the edit box contains text and the user clicks the button, the text is added to the list box. When the edit box is empty, the button is disabled. This is the code of the HideComp example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add (Edit1.Text);
end;

procedure TForm1.Edit1Change(Sender: TObject);
```

```
begin
  Button1.Enabled := Length (Edit1.Text) <> 0;
end;
```

I've listed these methods only to show you that in a form's code, you usually refer to the available components, defining their interactions. For this reason, it seems impossible to get rid of the fields corresponding to the component. However, you can hide them, moving them from the default published section to the private section of the form class declaration:

```
TForm1 = class(TForm)
  procedure Button1Click(Sender: TObject);
  procedure Edit1Change(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  Button1: TButton;
  Edit1: TEdit;
  ListBox1: TListBox;
end;
```

Now, if you run the program you'll get in trouble: The form will load, but because the private fields are not initialized, the events will use `nil` object references. Delphi usually initializes the published fields of the form using the components created from the DFM file. What if you do it yourself, with the following code?

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1 := FindComponent ('Button1') as TButton;
  Edit1 := FindComponent ('Edit1') as TEdit;
  ListBox1 := FindComponent ('ListBox1') as TListBox;
end;
```

It will *almost* work, but it generates a system error, similar to the one discussed in the previous section. This time, the private declarations will cause the linker to link in the implementations of those classes; the problem is, the streaming system needs to know the names of the classes in order to locate the class reference needed to construct the components while loading the DFM file.

The final touch you need is registration code to tell Delphi at run time about the existence of the component classes you want to use. You should do this before the form is created, so I generally place this code in the initialization section of the unit:

```
initialization
  RegisterClasses ([TButton, TEdit, TListBox]);
```

The question is, is this really worth the effort? You obtain a higher degree of encapsulation, protecting the components of a form from other forms (and other programmers writing them). Replicating these steps for every form can be tedious, so I wrote a wizard to generate the code for me on the fly. The wizard is far from perfect, because it doesn't handle changes automatically, but it is usable. See Appendix A, "Extra Delphi Tools by the Author," for more information about how to get it. For a large project built according to the principles of object-oriented programming, I recommend you consider this or a similar technique.

The Customizable Tag Property

The `Tag` property is strange, because it has no effect at all. It is merely an extra memory location, present in each component class, where you can store custom values. The kind of information stored and the way it is used are completely up to you.

It is often useful to have an extra memory location to attach information to a component without needing to define it in your component class. Technically, the `Tag` property stores a long integer so that, for example, you can store the entry number of an array or list that corresponds to an object. Using typecasting, you can store in the `Tag` property a pointer, an object reference, or anything else that is four bytes wide. A programmer can associate virtually anything with a component using its tag. You'll see how to use this property in several examples in future chapters.

Events

Now that I've discussed the `TComponent` class, I need to introduce one more element of Delphi. Delphi components are programmed using PME: properties, methods, and events. Methods and properties should be clear by now, but you haven't yet learned about events. The reason is that events don't imply a new language feature but are simply a standard coding technique. An event is technically a property—the only difference is that it refers to a method (a method pointer type, to be precise) instead of other types of data.

Events in Delphi

When a user does something with a component, such as click it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding operating system message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than operating system messages, and Delphi provides a number of extra inter-component messages.

From a theoretical point of view, an event is the result of a request sent to a component or control, which can respond to the message. Following this approach, to handle the click event of a button, you would need to subclass the `TButton` class and add the new event handler code inside the new class.

In practice, creating a new class for every component you want to use is too complex to be a reasonable solution. In Delphi, a component's event handler usually is a method of the form that holds the component, not of the component itself. In other words, the component relies on its owner, the form, to handle its events. This technique is called *delegation*, and it is fundamental to the Delphi component-based model. This way, you don't have to modify the `TButton` class, unless you want to define a new type of component, but can simply customize its owner to modify the behavior of the button.

NOTE *As you'll see in the next section, events in Delphi are based on pointers to methods. This is quite different from Java, which uses* listener *classes with methods for a family of events. These listener methods call the event handlers. C# and .NET use the similar idea of* delegate *classes, covered in Chapter 24, "The Microsoft .NET Architecture from the Delphi Perspective." Notice that the term* delegate *is the same traditionally used in the Delphi literature to explain the idea of event handlers.*

Method Pointers

Events rely on a specific feature of the Delphi language: *method pointers*. A method pointer type is like a procedural type, but one that refers to a method. Technically, a method pointer type is a procedural type that has an implicit `Self` parameter. In other words, a variable of a procedural type stores the address of a function to call, provided it has a given set of parameters. A method pointer variable stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as `Self` inside the method body when the method code is called using this method pointer.

NOTE *This explains the definition of Delphi's generic* `TMethod` *type, a record with a* `Code` *field and a* `Data` *field.*

The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords of `object` at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

When you have declared such a method pointer type, you can declare a variable of this type and assign to it a compatible method—a method that has the same signature (parameters, return type, calling convention)—of another object.

When you add an `OnClick` event handler for a button, Delphi does exactly that. The button has a method pointer type property, named `OnClick`, and you can directly or indirectly assign to it a method of another object, such as a form. When a user clicks the button, this method is executed, even if you have defined it inside another class.

What follows is a sketch of the code Delphi uses to define the event handler of a button component and the related method of a form:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;
```

Now, inside a procedure, you can write

```
MyButton.OnClick := Form1.Button1Click;
```

The only real difference between this code fragment and the VCL code is that `OnClick` is a property name, and the data it refers to is called `FOnClick`. An event that shows up in the Events page of the Object Inspector is nothing more than a property of a method pointer type. This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it.

Events Are Properties

I've already mentioned that events are properties. To handle an event of a component, you assign a method to the corresponding event property. When you double-click an event value in the Object Inspector, a new method is added to the owner form and assigned to the proper event property of the component.

It is possible for several events to share the same event handler or change an event handler at run time. To use this feature, you don't need much knowledge of the language. In fact, when you select an event in the Object Inspector, you can click the arrow button to the right of the event name to see a drop-down list of compatible methods—methods having the same signature of the method pointer type. Using the Object Inspector, it is easy to select the same method for the same event of different components or for different, compatible events of the same component.

Just as you added some properties to the `TDate` class in Chapter 2, you can add one event. The event will be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the date value has changed. To define an event, you simply define a property corresponding to it and add some data to store the method pointer the event refers to. These are the new definitions added to the class, available in the DateEvt example:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FOnChange write FOnChange;
    ...
  end;
```

The property definition is simple. A user of this class can assign a new value to the property and, hence, to the `FOnChange` private field. The class doesn't assign a value to this `FOnChange` field; the user of the component does the assignment. The `TDate` class simply calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned. The `DoChange` method (declared as a dynamic method as is traditional with event-firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

The `DoChange` method in turn is called every time one of the values changes, as in the following method:

```
procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
  // fire the event
  DoChange;
```

If you look at the program that uses this class, you can simplify its code considerably. First, add a new custom method to the form class:

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

The method's code simply updates the label with the current value of the `TDate` object's `Text` property:

```
procedure TDateForm.DateChange;
begin
  LabelDate.Caption := TheDay.Text;
end;
```

This event handler is then installed in the `FormCreate` method:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (2003, 7, 4);
  LabelDate.Caption := TheDay.Text;
  // assign the event handler for future changes
  TheDay.OnChange := DateChange;
end;
```

This seems like a lot of work. Was I lying when I told you the event handler would save you some coding? No. Now, after you've added some code, you can forget about updating the label when you change some of the object data. For example, here is the handler of the `OnClick` event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
end;
```

The same simplified code is present in many other event handlers. Once you have installed the event handler, you don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that you had to write some code at the beginning because this is not a component installed in Delphi but simply a class. With a component, you select the event handler in the Object Inspector and write a single line of code to update the label—that's all.

NOTE *This is just a short introduction to defining events. A basic understanding of these features is important for every Delphi programmer. If your aim is to write new components with complex events, you'll find a lot more information on all these topics in Chapter 9 ("Writing Delphi Components").*

Lists and Container Classes

It is often important to handle groups of components or objects. In addition to using standard arrays and dynamic arrays, a few VCL classes represent lists of other objects. These classes can be divided into three groups: simple lists, collections, and containers.

Lists and String Lists

Lists are represented by the generic list of objects, `TList`, and by the two lists of strings, `TStrings` and `TStringList`:

- `TList` defines a list of pointers, which can be used to store objects of any class. A `TList` is more flexible than a dynamic array, because it can be expanded automatically simply by adding new items to it. The advantage of a dynamic array over a `TList` is that the dynamic array allows you to indicate a specific type for contained objects and perform the proper compile-time type checking.
- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves, such as a list box.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with their own storage. You can use this class to define a list of strings in a program.

`TStringList` and `TStrings` objects have both a list of strings and a list of objects associated with the strings. These classes have a number of different uses. For example, you can use them for dictionaries of associated objects or to store bitmaps or other elements to be used in a list box.

The two classes of string lists also have ready-to-use methods to store or load their contents to or from a text file: `SaveToFile` and `LoadFromFile`. To loop through a list, you can use a simple `for` statement based on its index, as if the list were an array.

All these lists have a number of methods and properties. You can operate on lists using the array notation (`[` and `]`) both to read and to change elements. There is a `Count` property, as well as typical access methods, such as `Add`, `Insert`, `Delete`, `Remove`; search methods (for example, `IndexOf`); and sorting support. The `TList` class has an `Assign` method that, besides copying the source data, can perform set operations on the two lists, including *and*, *or*, and *xor*.

To fill a string list with items and later check whether one is present, you can write code like this:

```
var
  sl: TStringList;
  idx: Integer;
begin
  sl := TStringList.Create;
  try
```

```
      sl.Add ('one');
      sl.Add ('two');
      sl.Add ('three');
      // later
      idx := sl.IndexOf ('two');
      if idx >= 0 then
        ShowMessage ('String found');
    finally
      sl.Free;
    end;
  end;
```

NAME-VALUE PAIRS (AND DELPHI 7 EXTENSIONS)

The `TStringList` class has always had another nice feature: support for name-value pairs. If you add to a list a string like 'lastname=john', you can then search for the existence of the pair using the `IndexOfName` function or the `Values` array property. For example, you can retrieve the value 'john' by calling `Values ['lastname']`.

You can use this feature to build much more complex data structures, such as dictionaries, and still benefit from the possibility of attaching an object to the string. This data structure maps directly to initialization files and other common formats.

Delphi 7 further extends the possibilities of name-value pair support by allowing you to customize the separator, beyond the equal sign, using the new `NameValueSeparator` property. In addition, the new `ValueFromIndex` property gives you direct access to the value portion of a string at a given position; you no longer have to extract the name value manually from the complete string using a cumbersome (and extremely slow) expression:

```
str := MyStringList.Values [MyStringList.Names [I]]; // old
str := MyStringList.ValueFromIndex [I]; // new
```

USING LISTS OF OBJECTS

I've written an example focusing on the use of the generic `TList` class. When you need a list of any kind of data, you can generally declare a `TList` object, fill it with the data, and then access the data while casting it to the proper type. The ListDemo example demonstrates this approach and also shows its pitfalls. The form has a private variable, holding a list of dates:

```
private
  ListDate: TList;
```

This list object is created when the form itself is created:

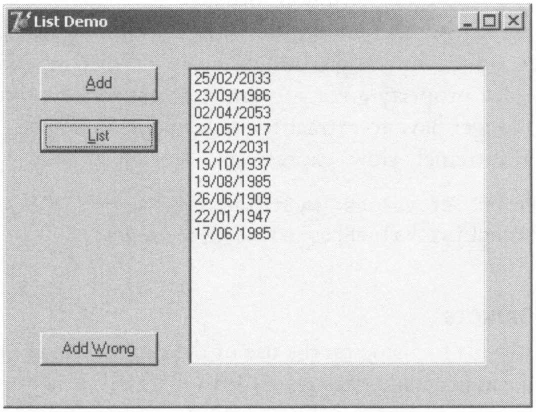
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ListDate := TList.Create;
end;
```

A button on the form adds a random date to the list (of course, I've included in the project the unit containing the date component built in the previous chapter):

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  ListDate.Add (TDate.Create (1900 + Random (200), 1 + Random (12),
    1 + Random (30)));
end;
```

When you extract the items from the list, you have to cast them back to the proper type, as in the following method, which is connected to the List button (you can see its effect in Figure 4.4):

```
procedure TForm1.ButtonListDateClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add ((TObject(ListDate [I]) as TDate).Text);
end;
```



```
25/02/2033
23/09/1986
02/04/2053
22/05/1917
12/02/2031
19/10/1937
19/08/1985
26/06/1909
22/01/1947
17/06/1985
```

FIGURE 4.4
The list of dates shown by the ListDemo example

At the end of the code, before you can do an `as` downcast, you first need to hard-cast the pointer returned by the `TList` into a `TObject` reference. This kind of expression can result in an invalid typecast exception, or it can generate a memory error when the pointer is not a reference to an object.

TIP *If there were no possibility of having anything but date objects in the list, extracting it with a static cast rather than an* `as` *cast would be more efficient. However, when there's even a remote chance of having a wrong object, I'd suggest using the* `as` *cast.*

To demonstrate that things can indeed go wrong, I've added one more button, which adds a `TButton` object to the list by calling `ListDate.Add(Sender)`. If you click this button and then update one of the lists, you'll get an error. Finally, remember that when you destroy a list of objects, you should destroy all of the list's objects first. The ListDemo program does this in the form's `FormDestroy` method:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ListDate.Count - 1 do
    TObject(ListDate [I]).Free;
  ListDate.Free;
end;
```

Collections

The second group, collections, contains only two VCL classes: `TCollection` and `TCollectionItem`. `TCollection` defines a homogeneous list of objects, which are owned by the collection class. The objects in the collection must be descendants of the `TCollectionItem` class. If you need a collection storing specific objects, you have to create both a subclass of `TCollection` and a matching subclass of `TCollectionItem`.

Collections are used to specify values of component properties; it is very unusual to work with collections to store your own objects. I discuss collections in Chapter 9.

Container Classes

Recent versions of Delphi include a series of container classes, defined in the Contnrs unit. The container classes extend the `TList` classes by adding the idea of ownership and by defining specific extraction rules (mimicking stacks and queues) or sorting capabilities.

The basic difference between `TList` and the new `TObjectList` class, for example, is that the latter is defined as a list of `TObject` objects, not a list of pointers. Even more important, however, is the fact that if the object list has the `OwnsObjects` property set to True, it automatically deletes an object when it is replaced by another one and deletes each object when the list itself is destroyed. Here's a list of the container classes:

- The `TObjectList` class (already described) represents a list of objects, eventually owned by the list itself.
- The inherited class `TComponentList` represents a list of components, with full support for destruction notification (an important safety feature when two components are connected using their properties; that is, when a component is the value of a property of another component).
- The `TClassList` class is a list of class references. It inherits from `TList` and requires no specific destruction, as you don't have to destroy class references in Delphi.
- The classes `TStack` and `TObjectStack` represent lists of pointers and objects, from which you can only extract elements starting from the last one you've inserted. A stack follows LIFO order (last in, first out). The typical methods of a stack are `Push` for insertion, `Pop` for extraction, and `Peek` to preview the first item without removing it. You can still use all the methods of the base class, `TList`.

- The classes `TQueue` and `TObjectQueue` represent lists of pointers and objects, from which you always remove the *first* item you've inserted (FIFO: first in, first out). The methods of these classes are the same as those of the stack classes, but they behave differently.

WARNING *Unlike* `TObjectList`*, the* `TObjectStack` *and* `TObjectQueue` *classes do not own the inserted objects and will not destroy those objects left in the data structure when it is destroyed. You can simply* `Pop` *all the items, destroy them once you're finished using them, and then destroy the container.*

To demonstrate the use of these classes, I've modified the earlier ListDate example into the Contain example. First, I changed the type of the `ListDate` variable to `TObjectList`. In the `FormCreate` method, I've modified the list creation to the following code, which activates the list ownership:

```
ListDate := TObjectList.Create (True);
```

At this point, you can simplify the destruction code, because applying `Free` to the list will automatically free the dates it holds.

I've also added to the program a stack and a queue object, filling each of them with numbers. One of the form's two buttons displays a list of the numbers in each container, and the other removes the last item (displayed in a message box):

```
procedure TForm1.btnQueueClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to Stack.Count - 1 do begin
    ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));
    Queue.Push(Queue.Pop);
  end;
  ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));
end;
```

By clicking the two buttons, you can see that calling `Pop` for each container returns the last item. The difference is that the `TQueue` class inserts elements at the beginning, and the `TStack` class inserts them at the end.

HASHED ASSOCIATIVE LISTS

Since Delphi 6, the set of predefined container classes includes `TBucketList` and `TObjectBucketList`. These two lists are associative, which means they have a key and an actual entry. The key is used to identify the items and search for them. To add an item, you call the `Add` method with two parameters: the key and the data. When you use the `Find` method, you pass the key and retrieve the data. The same effect is achieved by using the `Data` array property, passing the key as parameter.

These lists are based on a hash system. The lists create an internal array of items, called *buckets*, each having a sublist of list elements. As you add an item, its key value is used to compute the *hash* value, which determines the bucket to which the item should be added. When searching the item, the hash is computed again, and the list immediately grabs the sublist containing the item and searches for it there. This process makes for very fast insertion and searches, but only if the hash algorithm distributes the items evenly among the various buckets and if there are enough different entries in the array. When many elements can be in the same bucket, searching is slower.

For this reason, as you create the `TObjectBucketList`, you can specify the number of entries for the list by using the parameter of the constructor and choosing a value between 2 and 256. The value of the bucket is determined by taking the first byte of the pointer (or number) passed as the key and doing an `and` operation with a number corresponding to the entries.

NOTE *I don't find this algorithm very convincing for a hash system, but replacing it with your own implies only overriding the* `BucketFor` *virtual function and eventually changing the number of entries in the array, by setting a different value for the* `BucketCount` *property.*

Another interesting feature, not available for lists, is the `ForEach` method, which allows you to execute a given function on each item contained in the list. You pass the `ForEach` method a pointer to your data and a procedure that receives four parameters: your custom pointer, each key and object of the list, and a Boolean parameter you can set to False to stop the execution. In other words, these are the two signatures:

```
type
  TBucketProc = procedure(AInfo, AItem, AData: Pointer;
    out AContinue: Boolean);

function TCustomBucketList.ForEach(AProc: TBucketProc;
  AInfo: Pointer): Boolean;
```

NOTE *In addition to these containers, Delphi includes a* `THashedStringList` *class, which inherits from* `TStringList`*. This class has no direct relationship with the hashed lists and is defined in a different unit, IniFiles. The hashed string list has two associated hash tables (of type* `TStringHash`*), which are completely refreshed every time the content of the string list changes. So, this class is useful only for reading a large set of fixed strings, not for handling a list of strings changing often over time. On the other hand, the* `TStringHash` *support class seems to be quite useful in general cases, and has a good algorithm for computing the hash value of a string.*

Type-Safe Containers and Lists

Containers and lists have a problem: They are not type-safe, as I've shown in both examples by adding a button object to a list of dates. To ensure that the data in a list is homogenous, you can check the type of the data you extract before you insert it, but as an extra safety measure you might also want to check the type of the data while extracting it. However, adding run-time type checking slows a program and is risky—a programmer might fail to check the type in some cases.

To solve both problems, you can create specific list classes for given data types and fashion the code from the existing `TList` or `TObjectList` class (or another container class). There are two approaches to accomplish this:

- Derive a new class from the list class and customize the `Add` method and the access methods, which relate to the `Items` property. This is also the approach used by Borland for the container classes, which all derive from `TList`.

NOTE *Delphi container classes use static overrides to perform simple type conveniences (parameters and function results of the desired type). Static overrides are not the same as polymorphism; someone using a container class via a* `TList` *variable will not be calling the container's specialized functions. Static override is a simple and effective technique, but it has one very important restriction: The methods in the descendent should not do anything beyond simple typecasting, because you have no guarantee the descendent methods will be called. The list might be accessed and manipulated using the ancestor methods as much as by the descendent methods, so their operations must be identical. The only difference is the type used in the descendent methods, which allows you to avoid extra typecasting.*

- Create a brand-new class that contains a `TList` object, and map the methods of the new class to the internal list using proper type checking. This approach defines a wrapper class, a class that "wraps" around an existing one to provide a different or limited access to its methods (in our case, to perform a type conversion).

I've implemented both solutions in the DateList example, which defines lists of `TDate` objects. In the code that follows, you'll find the declaration of the two classes, the inheritance-based `TDateListI` class and the wrapper class `TDateListW`:

```
type
  // inheritance-based
  TDateListI = class (TObjectList)
  protected
    procedure SetObject (Index: Integer; Item: TDate);
    function GetObject (Index: Integer): TDate;
  public
    function Add (Obj: TDate): Integer;
    procedure Insert (Index: Integer; Obj: TDate);
    property Objects [Index: Integer]: TDate
      read GetObject write SetObject; default;
  end;

  // wrapper based
  TDateListW = class(TObject)
  private
    FList: TObjectList;
    function GetObject (Index: Integer): TDate;
    procedure SetObject (Index: Integer; Obj: TDate);
    function GetCount: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    function Add (Obj: TDate): Integer;
    function Remove (Obj: TDate): Integer;
    function IndexOf (Obj: TDate): Integer;
    property Count: Integer read GetCount;
    property Objects [Index: Integer]: TDate
      read GetObject write SetObject; default;
  end;
```

Obviously, the first class is simpler to write—it has fewer methods, and they just call the inherited ones. The good thing is that a `TDateListI` object can be passed to parameters expecting a `TList`. The problem is that the code that manipulates an instance of this list via a generic `TList` variable will not be calling the specialized methods, because they are not virtual and might end up adding to the list objects of other data types.

Instead, if you decide not to use inheritance, you end up writing a lot of code; you need to reproduce every one of the original `TList` methods, simply calling the methods of the internal `FList` object. The drawback is that the `TDateListW` class is not type compatible with `TList`, which limits its usefulness. It can't be passed as parameter to methods expecting a `TList`.

Both of these approaches provide good type checking. After you've created an instance of one of these list classes, you can add only objects of the appropriate type, and the objects you extract will naturally be of the correct type. This technique is demonstrated by the DateList example. This program has a few buttons, a combo box to let a user choose which of the lists to show, and a list box to show the list values. The program stretches the lists by trying to add a button to the list of `TDate` objects. To add an object of a different type to the `TDateListI` list, you can convert the list to its base class, `TList`. This might happen accidentally if you pass the list as a parameter to a method that expects an ancestor class of the list class. In contrast, for the `TDateListW` list to fail, you must explicitly cast the object to `TDate` before inserting it, something a programmer should never do:

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);
begin
  ListW.Add (TDate(TButton.Create (nil)));
  TList(ListI).Add (TButton.Create (nil));
  UpdateList;
end;
```

The `UpdateList` call triggers an exception, displayed directly in the list box, because I've used an `as` typecast in the custom list classes. A wise programmer should never write the previous code. To summarize, writing a custom list for a specific type makes a program much more robust. Writing a wrapper list instead of one that's based on inheritance tends to be a little safer, although it requires more coding.

NOTE *Instead of rewriting wrapper-style list classes for different types, you can use my List Template Wizard. See Appendix A for details.*

Streaming

Another core area of the Delphi class library is its support for streaming, which includes file management, memory, sockets, and other sources of information arranged in a sequence. The idea of streaming is that you move through the data while reading it, much like the `Read` and `Write` functions traditionally used by the Pascal language (and discussed in Chapter 12 of *Essential Pascal* (see Appendix C for availability of this e-book).

The *TStream* Class

The VCL defines the abstract `TStream` class and several subclasses. The parent class, `TStream`, has just a few properties, and you'll never create an instance of it, but it has an interesting list of methods you'll generally use when working with derived stream classes.

The `TStream` class defines two properties, `Size` and `Position`. All stream objects have a specific size (which generally grows if you write something after the end of the stream), and you must specify a position within the stream where you want to either read or write information.

Reading and writing bytes depends on the actual stream class you are using, but in both cases you don't need to know much more than the size of the stream and your relative position in the stream to read or write data. In fact, that's one of the advantages of using streams. The basic interface remains the same whether you're manipulating a disk file, a binary large object (BLOB) field, or a long sequence of bytes in memory.

In addition to the `Size` and `Position` properties, the `TStream` class also defines several important methods, most of which are virtual and abstract. (In other words, the `TStream` class doesn't define what these methods do; therefore, derived classes are responsible for implementing them.) Some of these methods are important only in the context of reading or writing components within a stream (for instance, `ReadComponent` and `WriteComponent`), but some are useful in other contexts, too. In Listing 4.2, you can find the declaration of the `TStream` class, extracted from the Classes unit.

LISTING 4.2: THE PUBLIC PORTION OF THE DEFINITION OF THE *TSTREAM* CLASS

```
TStream = class(TObject)
public
  // read and write a buffer
  function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
  function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
  procedure ReadBuffer(var Buffer; Count: Longint);
  procedure WriteBuffer(const Buffer; Count: Longint);

  // move to a specific position
  function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;
  function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
    overload; virtual;

  // copy the stream
  function CopyFrom(Source: TStream; Count: Int64): Int64;

  // read or write a component
  function ReadComponent(Instance: TComponent): TComponent;
  function ReadComponentRes(Instance: TComponent): TComponent;
  procedure WriteComponent(Instance: TComponent);
  procedure WriteComponentRes(const ResName: string; Instance: TComponent);
  procedure WriteDescendent(Instance, Ancestor: TComponent);
  procedure WriteDescendentRes(
    const ResName: string; Instance, Ancestor: TComponent);
  procedure WriteResourceHeader(const ResName: string; out FixupInfo: Integer);
  procedure FixupResourceHeader(FixupInfo: Integer);
  procedure ReadResHeader;
  // properties
  property Position: Int64 read GetPosition write SetPosition;
  property Size: Int64 read GetSize write SetSize64;
end;
```

The basic use of a stream involves calling the `ReadBuffer` and `WriteBuffer` methods, which are very powerful but not terribly easy to use. The first parameter is an untyped buffer in which you can pass the variable to save from or load to. For example, you can save into a file a number (in binary format) and a string, with this code:

```
var
  stream: TStream;
  n: integer;
  str: string;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test', fmCreate);
  stream.WriteBuffer (n, sizeOf(integer));
  stream.WriteBuffer (str[1], Length (str));
  stream.Free;
```

An alternative approach is to let specific components save or load data to and from streams. Many VCL classes define a `LoadFromStream` or a `SaveToStream` method, including `TStrings`, `TStringList`, `TBlobField`, `TMemoField`, `TIcon`, and `TBitmap`.

Specific Stream Classes

Creating a `TStream` instance makes no sense, because this class is abstract and provides no direct support for saving data. Instead, you can use one of the derived classes to load data from or store it to an actual file, a BLOB field, a socket, or a memory block. Use `TFileStream` when you want to work with a file, passing the filename and some file access options to the `Create` method. Use `TMemoryStream` to manipulate a stream in memory and not an actual file.

Several units define `TStream`-derived classes. The Classes unit includes the following classes:

- `THandleStream` defines a stream that manipulates a disk file represented by a file handle.
- `TFileStream` defines a stream that manipulates a disk file (a file that exists on a local or network disk) represented by a filename. It inherits from `THandleStream`.
- `TCustomMemoryStream` is the base class for streams stored in memory but is not used directly.
- `TMemoryStream` defines a stream that manipulates a sequence of bytes in memory. It inherits from `TCustomMemoryStream`.
- `TStringStream` provides a simple way to associate a stream to a string in memory, so that you can access the string with the `TStream` interface and also copy the string to and from another stream.
- `TResourceStream` defines a stream that manipulates a sequence of bytes in memory, and provides read-only access to resource data linked into the executable file of an application (the DFM files are an example of this resource data). It inherits from `TCustomMemoryStream`.

Stream classes defined in other units include the following:

- `TBlobStream` defines a stream that provides simple access to database BLOB fields. There are similar BLOB streams for database access technologies other than the BDE, including `TSQLBlobStream` and `TClientBlobStream`. (Notice that each type of dataset uses a specific stream class for BLOB fields.) All these classes inherit from `TMemoryStream`.
- `TOleStream` defines a stream for reading and writing information over the interface for streaming provided by an OLE object.
- `TWinSocketStream` provides streaming support for a socket connection.

Using File Streams

Creating and using a file stream can be as simple as creating a variable of a type that descends from `TStream` and calling components' methods to load content from the file:

```
var
  S: TFileStream;
begin
  if OpenDialog1.Execute then
  begin
    S := TFileStream.Create (OpenDialog1.FileName, fmOpenRead);
    try
      Memo1.Lines.LoadFromStream (S);
    finally
      S.Free;
    end;
  end;
end;
```

As you can see in this code, the `Create` method for file streams has two parameters: the name of the file and a flag indicating the requested access mode. In this case, you want to read the file, so you use the `fmOpenRead` flag (other available flags are documented in the Delphi help).

NOTE *Of the different modes, the most important are fmShareDenyWrite, which you'll use when you're simply reading data from a shared file, and fmShareExclusive, which you'll use when you're writing data to a shared file. There is a third parameter in* `TFileStream.Create`*, called* `Rights`*. This parameter is used to pass file access permissions to the Linux filesystem when the access mode is fmCreate (that is, only when you are creating a new file). This parameter is ignored on Windows.*

A big advantage of streams over other file access techniques is that they're very interchangeable, so you can work with memory streams and then save them to a file, or you can perform the opposite operations. This might be a way to improve the speed of a file-intensive program. Here is a snippet of a file-copying function to give you another idea of how you can use streams:

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
```

```
begin
  Stream1 := TFileStream.Create (SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create (TargetName, fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom (Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end
  finally
    Stream1.Free;
  end
end;
```

Another important use of streams is to handle database BLOB fields or other large fields directly. You can export such data to a stream or read it from one by calling the `SaveToStream` and `LoadFromStream` methods of the `TBlobField` class.

NOTE *Delphi 7 streaming support adds a new exception base class,* `EFileStreamError`*. Its constructor takes as parameter a filename for error reporting. This class standardizes and largely simplifies the notification of file-related errors in streams.*

The *TReader* and *TWriter* Classes

By themselves, the VCL stream classes don't provide much support for reading or writing data. In fact, stream classes don't implement much beyond simply reading and writing blocks of data. If you want to load or save specific data types in a stream (and don't want to perform a great deal of typecasting), you can use the `TReader` and `TWriter` classes, which derive from the generic `TFiler` class.

Basically, the `TReader` and `TWriter` classes exist to simplify loading and saving stream data according to its type, and not just as a sequence of bytes. To do this, `TWriter` embeds special signatures into the stream that specify the type for each object's data. Conversely, the `TReader` class reads these signatures from the stream, creates the appropriate objects, and then initializes those objects using the subsequent data from the stream.

For example, I could have written out a number and a string to a stream by writing:

```
var
  stream: TStream;
  n: integer;
  str: string;
  w: TWriter;
begin
  n := 10;
  str := 'test string';
  stream := TFileStream.Create ('c:\tmp\test.txt', fmCreate);
  w := TWriter.Create (stream, 1024);
  w.WriteInteger (n);
  w.WriteString (str);
  w.Free;
  stream.Free;
```

This time the file will include the extra signature characters, so I can read back this file only by using a `TReader` object. For this reason, using `TReader` and `TWriter` is generally confined to component streaming and is seldom applied in general file management.

Streams and Persistency

In Delphi, streams play a considerable role in persistency. For this reason, many methods of `TStream` relate to saving and loading a component and its subcomponents. For example, you can store a form in a stream by writing

```
stream.WriteComponent(Form1);
```

If you examine the structure of a Delphi DFM file, you'll discover that it's really just a resource file that contains a custom format resource. Inside this resource, you'll find the component information for the form or data module and for each of the components it contains. As you would expect, the stream classes provide two methods to read and write this custom resource data for components: `WriteComponentRes` to store the data, and `ReadComponentRes` to load it.

For your experiment in memory (not involving DFM files), though, using `WriteComponent` is generally better suited. After you create a memory stream and save the current form to it, the problem is how to display it. You can do this by transforming the form's binary representation to a textual representation. Even though the Delphi IDE, since version 5, can save DFM files in text format, the representation used internally for the compiled code is invariably a binary format.

The IDE can accomplish the form conversion, generally with the View as Text command of the Form Designer, and in other ways. The Delphi `Bin` directory also contains a command-line utility, `CONVERT.EXE`. Within your own code, the standard way to obtain a conversion is to call the specific VCL methods. There are four functions for converting to and from the internal object format obtained by the `WriteComponent` method:

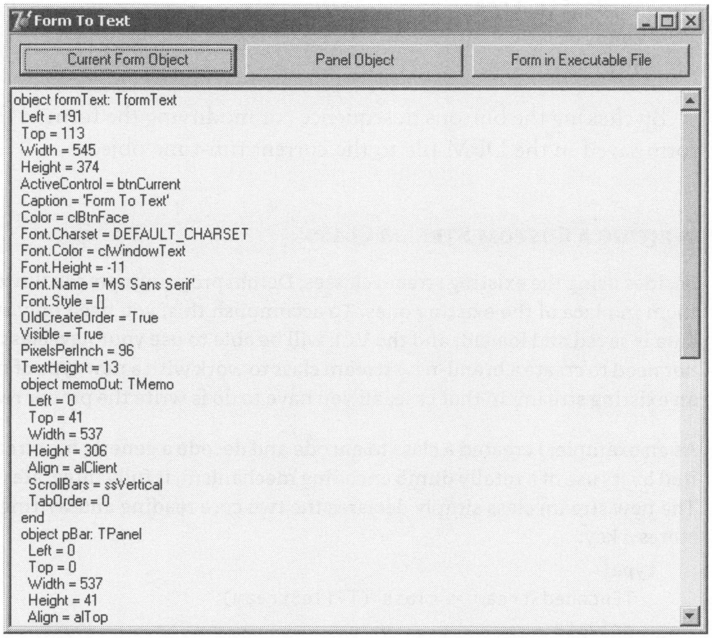
```
procedure ObjectBinaryToText(Input, Output: TStream); overload;
procedure ObjectBinaryToText(Input, Output: TStream;
  var OriginalFormat: TStreamOriginalFormat); overload;
procedure ObjectTextToBinary(Input, Output: TStream); overload;
procedure ObjectTextToBinary(Input, Output: TStream;
  var OriginalFormat: TStreamOriginalFormat); overload;
```

Four different functions, with the same parameters and names containing the name *Resource* instead of *Binary* (as in `ObjectResourceToText`), convert the resource format obtained by `WriteComponentRes`. A final method, `TestStreamFormat`, indicates whether a DFM is storing a binary or textual representation.

In the FormToText program, I've used the `ObjectBinaryToText` method to copy the binary definition of a form into another stream, and then I've displayed the resulting stream in a memo, as you can see in Figure 4.5. Here is the code of the two methods involved:

```
procedure TformText.btnCurrentClick(Sender: TObject);
var
  MemStr: TStream;
begin
  MemStr := TMemoryStream.Create;
  try
```

```
    MemStr.WriteComponent (Self);
    ConvertAndShow (MemStr);
  finally
    MemStr.Free
  end;
end;

procedure TformText.ConvertAndShow (aStream: TStream);
var
  ConvStream: TStream;
begin
  aStream.Position := 0;
  ConvStream := TMemoryStream.Create;
  try
    ObjectBinaryToText (aStream, ConvStream);
    ConvStream.Position := 0;
    MemoOut.Lines.LoadFromStream (ConvStream);
  finally
    ConvStream.Free
  end;
end;
```



```
object formText: TformText
  Left = 191
  Top = 113
  Width = 545
  Height = 374
  ActiveControl = btnCurrent
  Caption = 'Form To Text'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = True
  Visible = True
  PixelsPerInch = 96
  TextHeight = 13
  object memoOut: TMemo
    Left = 0
    Top = 41
    Width = 537
    Height = 306
    Align = alClient
    ScrollBars = ssVertical
    TabOrder = 0
  end
  object pBar: TPanel
    Left = 0
    Top = 0
    Width = 537
    Height = 41
    Align = alTop
```

FIGURE 4.5
The textual description of a form component, displayed inside itself by the FormToText example

Notice that by repeatedly clicking the Current Form Object button you'll get more and more text, and the text of the memo is included in the stream. After a few times, the entire operation will become extremely slow, until the program seems to be hung up. In this code, you see some of the flexibility of using streams—you can write a generic procedure that you can use to convert any stream.

NOTE *It's important to stress that after you've written data to a stream, you must explicitly seek back to the beginning (or set the* `Position` *property to 0) before you can use the stream further—unless you want to append data to the stream, of course.*

Another button, labeled Panel Object, shows the textual representation of a specific component, the panel, passing the component to the `WriteComponent` method. The third button, Form in Executable File, performs a different operation. Instead of streaming an existing object in memory, it loads in a `TResourceStream` object the design-time representation of the form—that is, its DFM file—from the corresponding resource embedded in the executable file:

```
procedure TFormText.btnResourceClick(Sender: TObject);
var
  ResStr: TResourceStream;
begin
  ResStr := TResourceStream.Create(hInstance, 'TFORMTEXT', RT_RCDATA);
  try
    ConvertAndShow (ResStr);
  finally
    ResStr.Free
  end;
end;
```

By clicking the buttons in sequence (or modifying the form of the program) you can compare the form saved in the DFM file to the current run-time object.

WRITING A CUSTOM STREAM CLASS

Besides using the existing stream classes, Delphi programmers can write their own stream classes and use them in place of the existing ones. To accomplish this, you need only specify how a generic block of raw data is saved and loaded, and the VCL will be able to use your new class wherever you call for it. You may not need to create a brand-new stream class to work with a new type of media, but only need to customize an existing stream. In that case, all you have to do is write the proper read and write methods.

As an example, I created a class to encode and decode a generic file stream. Although this example is limited by its use of a totally dumb encoding mechanism, it fully integrates with the VCL and works properly. The new stream class simply declares the two core reading and writing methods and has a property that stores a key:

```
type
  TEncodedStream = class (TFileStream)
  private
     FKey: Char;
  public
```

Continued on next page

WRITING A CUSTOM STREAM CLASS *(continued)*

```
    constructor Create(const FileName: string; Mode: Word);
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    property Key: Char read FKey write FKey;
  end;
```

The value of the key is added to each of the bytes saved to a file and subtracted when the data is read. Here is the complete code of the `Write` and Read methods, which uses pointers quite heavily:

```
constructor TEncodedStream.Create( const FileName: string; Mode: Word);
begin
  inherited Create (FileName, Mode);
  FKey := 'A'; // default
end;

function TEncodedStream.Write(const Buffer; Count: Longint): Longint;
var
  pBuf, pEnc: PChar;
  I, EncVal: Integer;
begin
  // allocate memory for the encoded buffer
  GetMem (pEnc, Count);
  try
    // use the buffer as an array of characters
    pBuf := PChar (@Buffer);
    // for every character of the buffer
    for I := 0 to Count - 1 do
    begin
      // encode the value and store it
      EncVal := ( Ord (pBuf[I]) + Ord(Key) ) mod 256;
      pEnc [I] := Chr (EncVal);
    end;
    // write the encoded buffer to the file
    Result := inherited Write (pEnc^, Count);
  finally
    FreeMem (pEnc, Count);
  end;
end;

function TEncodedStream.Read(var Buffer; Count: Longint): Longint;
var
  pBuf, pEnc: PChar;
  I, CountRead, EncVal: Integer;
begin
```

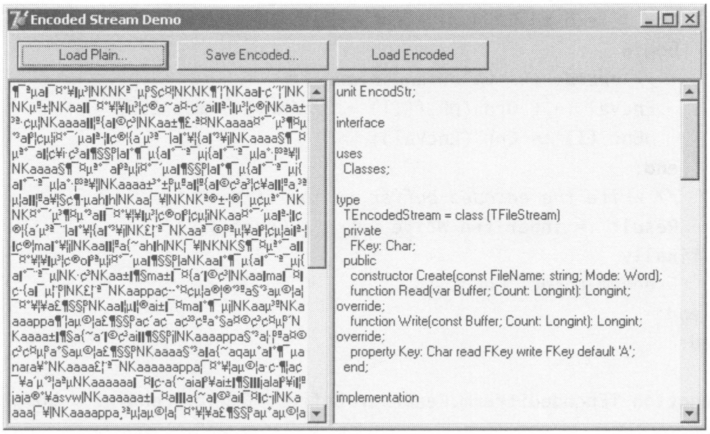
Continued on next page

WRITING A CUSTOM STREAM CLASS *(continued)*

```
    // allocate memory for the encoded buffer
    GetMem (pEnc, Count);
    try
      // read the encoded buffer from the file
      CountRead := inherited Read (pEnc^, Count);
      // use the output buffer as a string
      pBuf := PChar (@Buffer);
      // for every character actually read
      for I := 0 to CountRead - 1 do
      begin
        // decode the value and store it
        EncVal := ( Ord (pEnc[I]) - Ord(Key) ) mod 256;
        pBuf [I] := Chr (EncVal);
      end;
    finally
      FreeMem (pEnc, Count);
    end;
    // return the number of characters read
    Result := CountRead;
  end;
```

The comments in this rather complex code should help you understand the details.

Now I used this encoded stream in a demo program, called EncDemo. The form of this program has two memo components and three buttons, as you can see in the following graphic:



```
unit EncodStr;

interface

uses
  Classes;

type
  TEncodedStream = class (TFileStream)
  private
    FKey: Char;
  public
    constructor Create(const FileName: string; Mode: Word);
    function Read(var Buffer; Count: Longint): Longint;
override;
    function Write(const Buffer; Count: Longint): Longint;
override;
    property Key: Char read FKey write FKey default 'A';
  end;

implementation
```

The first button loads a plain text file in the first memo; the second button saves the text of this first memo in an encoded file; and the third button reloads the encoded file into the second memo, decoding it. In this example, after encoding the file, I've reloaded it in the first memo as a plain text file on the left, which of course is unreadable.

Continued on next page

WRITING A CUSTOM STREAM CLASS *(continued)*

Because the encoded stream class is available, the code of this program is very similar to that of any other program using streams. For example, here is the method used to save the encoded file (you can compare its code to that of earlier examples based on streams):

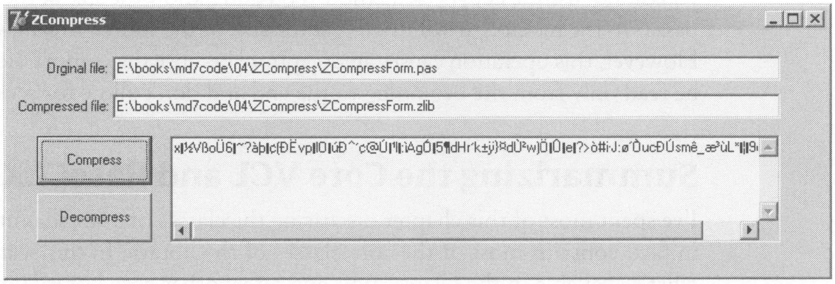
```
procedure TFormEncode.BtnSaveEncodedClick(Sender: TObject);
var
  EncStr: TEncodedStream;
begin
  if SaveDialog1.Execute then
  begin
    EncStr := TEncodedStream.Create(SaveDialog1.Filename, fmCreate);
    try
      Memo1.Lines.SaveToStream (EncStr);
    finally
      EncStr.Free;
    end;
  end;
end;
```

Compressing Streams with ZLib

A new feature of Delphi 7 is official support for the ZLib compression library (available and described at `www.gzip.org/zlib`). A unit interfacing ZLib has been available for a long time on Delphi's CD, but now it is included in the core distribution and is part of the VCL source (the ZLib and ZLibConst units). In addition to providing an interface to the library (which is a C library you can directly embed in the Delphi program, with no need to distribute a DLL), Delphi 7 defines a couple of helper stream classes: `TCompressStream` and `TDecompressStream`.

As an example of using these classes, I've written a small program called ZCompress that compresses and decompresses files. The program has two edit boxes in which you enter the name of the file to compress and the name of the resulting file, which is created if it doesn't already exist. When you click the Compress button, the source file is used to create the destination file; clicking the Decompress button moves the compressed file back to a memory stream. In both cases, the result of the compression or decompression is displayed in a memo. Figure 4.6 shows the result for the compressed file (which happens to be the source code of the form of the current program).

FIGURE 4.6 The ZCompress example can compress a file using the ZLib library.



Orginal file: E:\books\md7code\04\ZCompress\ZCompressForm.pas

Compressed file: E:\books\md7code\04\ZCompress\ZCompressForm.zlib

```
x|½VßoÜ6|~?àþ|ç(ÐËvp|lO|úÐ^´ç@Ú|¹|:ìAgÓ|5¶dHr´k±ÿ}¤dÙ²w)Ö|Ô|e|?>ò#i·J:ø´ÒucÐÚsmê_æ³ùL*|||9
```

To make the code of this program more reusable, I've written two functions for compressing or decompressing a stream into another stream. Here is the code:

```
procedure CompressStream (aSource, aTarget: TStream);
var
  comprStream: TCompressionStream;
begin
  comprStream := TCompressionStream.Create(
    clFastest, aTarget);
  try
    comprStream.CopyFrom(aSource, aSource.Size);
    comprStream.CompressionRate;
  finally
    comprStream.Free;
  end;
end;

procedure DecompressStream (aSource, aTarget: TStream) ;
var
  decompStream: TDecompressionStream;
  nRead: Integer;
  Buffer: array [0..1023] of Char;
begin
  decompStream := TDecompressionStream.Create(aSource);
  try
    // aStreamDest.CopyFrom (decompStream, size) doesn't work
    // properly as you don't know the size in advance,
    // so I've used a similar "manual" code
    repeat
      nRead := decompStream.Read(Buffer, 1024);
      aTarget.Write (Buffer, nRead);
    until nRead = 0;
  finally
    decompStream.Free;
  end;
end;
```

As you can see in the code comment, the decompression operation is slightly more complex because you cannot use the `CopyFrom` method: You don't know the size of the resulting stream in advance. If you pass 0 to the method, it will try to get the size of the source stream, which is a `TDecompressionStream`. However, this operation causes an exception, because the compression and decompression streams can be read only from the beginning to the end and don't allow for seeking the end of the file.

Summarizing the Core VCL and BaseCLX Units

I've spent most of this chapter discussing the classes of a single unit of the library: Classes. This unit, in fact, contains most of the core classes of the library. In this section, I'll provide an overview of what's available in the Classes unit and a few other core library units.

The Classes Unit

The Classes unit is at the heart of both VCL and CLX libraries, and although it has many internal changes from the last version of Delphi, little is new for average users. (Most changes are related to modified IDE integration and are meant for expert component writers.)

Here is a list of what you can find in the Classes unit, a unit that every Delphi programmer should spend some time with:

- Many enumerated types, the standard method pointer types (including `TNotifyEvent`), and many exception classes.
- Core library classes, including `TPersistent` and `TComponent` as well as many others seldom used directly.
- List classes, including `TList`, `TThreadList` (a thread-safe version of the list), `TInterfaceList` (a list of interfaces, used internally), `TCollection`, `TCollectionItem`, `TOwnedCollection` (which is simply a collection with an owner), `TStrings`, and `TStringList`.
- All the stream classes I discussed in the previous section but won't list here again. There are also the `TFiler`, `TReader`, and `TWriter` classes and a `TParser` class used internally for DFM parsing.
- Utility classes, such as `TBits` for binary manipulation and a few utility routines (for example, point and rectangle constructors, and string list manipulation routines such as `LineStart` and `ExtractStrings`). There are also many registration classes, to notify the system of the existence of components, classes, special utility functions you can replace, and much more.
- The `TDataModule` class, a simple object container alternative to a form. Data modules can contain only nonvisual components and are generally used in database and web applications.

NOTE *In early versions of Delphi, the* `TDataModule` *class was defined in the Forms unit; since Delphi 6 it has been moved to the Classes unit. This was done to eliminate the code overhead of the GUI classes from non-visual applications (for example, web server modules) and to better separate non-portable Windows code from OS-independent classes, such as* `TDataModule`. *Other changes relate to the data modules—for example, to allow the creation of web applications with multiple data modules.*

- New interface-related classes, such as `TInterfacedPersistent`, aimed at providing further support for interfaces. This particular class allows Delphi code to hold on to a reference to a `TPersistent` object or any descendent implementing interfaces, and is a core element of the new support for interfaced objects in the Object Inspector (see Chapter 9, "Writing Delphi Components," for an example).
- The new `TRecall` class, used to maintain a temporary copy of an object. This class is particularly useful for graphical-based resources.
- The new `TClassFinder` class, which is used to find a registered class instead of the `FindClass` method.
- The `TThread` class, which provides the core operating system–independent support for multithreaded applications.

NEW IN THE CLASSES UNIT

In Delphi 7, the Classes unit has only a few minor additions. Beside the changes I've already mentioned in this chapter, such as the extended support for name-value pairs in the `TStringList` class, there are a couple of new global functions, `AncestorIsValid` and `IsDefaultPropertyValue`.

Both functions were introduced to support the highlighting of non-default properties in the Object Inspector. They serve little other purpose, and I doubt you'll benefit from their use in an application—unless you are interested in saving the status of a component and form, and writing your own custom streaming mechanism.

Other Core Units

Typical Delphi programmers don't directly use the other units that are part of the RTL package as often as they use Classes. Here is a list of these other units:

- The TypInfo unit includes support for accessing RTTI information for published properties, as discussed in the section "Accessing Properties by Name."
- The SyncObjs unit contains a few generic classes for thread synchronization.
- The ZLib unit includes compression and decompression streams, as discussed earlier in the section "Compressing Streams with ZLib."
- The ObjAuto unit contains code to call the published methods of an object by name, passing the parameters in a variant array. This unit is part of the extended support for dynamic method invocation pushed by SOAP and other new Delphi technologies.

Of course, the RTL package also includes the units with functions and procedures discussed in the preceding chapter, such as Math, SysUtils, Variants, VarUtils, StrUtils, DateUtils, and so on.

What's Next?

As you have seen in this chapter, the Delphi class library has a few root classes that play a considerable role and that you should learn to leverage to the maximum possible extent. Some programmers tend to become expert on the components they use every day, and this is important; but without understanding the core classes (and ideas such as ownership and streaming), you'll have a tough time grasping the full power of Delphi.

Of course, in this book, I also need to discuss visual and database classes. Now that I've introduced all the base elements of Delphi (language, RTL, core classes), I'm ready to discuss the development of real applications with this tool. Chapter 5 covers the structure of the *visual* portion of the component library.

Following chapters are devoted to examples of using the various components to build applications with a modern user interface and use forms effectively. I'll cover the advanced use of traditional controls and menus, discuss the actions architecture, cover the `TForm` class, and then examine toolbars, status bars, dialog boxes, and MDI applications.

Chapter 5

Visual Controls

NOW THAT YOU'VE BEEN introduced to the Delphi environment and have seen an overview of the Delphi language and the base elements of component library, we are ready to delve into the use of components and the development of the user interface of applications. This is really what Delphi is about. Visual programming using components is a key feature of this development environment.

Delphi comes with a large number of ready-to-use components. I won't describe every component in detail, examining each of its properties and methods; if you need this information, you can find it in the Help system. The aim of this chapter and the following ones of this book is to show you how to use some of the advanced features offered by the Delphi predefined components to build applications and to discuss specific programming techniques.

I'll start with a comparison of the VCL and VisualCLX libraries and coverage of the core classes (particularly `TControl`). Then I'll examine the various visual components, because choosing the right basic controls will often help you get a project underway more quickly.

This chapter covers the following topics:

- VCL versus VisualCLX
- `TControl`, `TWinControl`, and `TWidgetControl`
- An overview of the standard components
- Basic and advanced menu construction
- Modifying the system menu
- Graphics in menus and list boxes
- Owner-draw and styles

VCL versus VisualCLX

As you saw in Chapter 4, "Core Library Classes," Delphi has two visual class libraries: the cross-platform library (CLX) alongside the traditional Windows library (VCL). There are certainly many differences, even in the use of the RTL and code library classes, between developing programs specifically for Windows or with a cross-platform attitude, but the differences are most striking in the user interface portion.

The visual portion of VCL is a wrapper of the Window API. It includes wrappers of the native Windows controls (like buttons and edit boxes), the common controls (like TreeViews and ListViews), plus a bunch of native Delphi controls bound to the Windows concept of a window. In addition, a `TCanvas` class wraps the basic graphic calls, so you can easily paint on the surface of a window.

VisualCLX, the visual portion of CLX, is a wrapper of the Qt (pronounced "cute") library. It includes wrappers of the native Qt widgets, which range from basic to advanced controls, very similar to Windows' standard and common controls. It also includes painting support using another, similar, `TCanvas` class. Qt is a C++ class library developed by Trolltech (`www.trolltech.com`), a Norwegian company with a strong relationship with Borland.

On Linux, Qt is one of the de facto standard user-interface libraries and is the basis of the KDE desktop environment. On Windows, Qt provides an alternative to the use of the native APIs. Unlike VCL, which provides a wrapper to the native controls, Qt provides an alternate implementation to those controls. Even if each of them is based on Windows's window, a QT button isn't a Windows BUTTON class control (you can see this by running WinSight32). This allows programs to be truly portable, because no hidden differences are created by the operating system (or introduced by the operating system vendor behind the scenes). It also allows you to avoid an extra layer; CLX on top of Qt on top of Windows native controls suggests three layers, but in fact there are two layers in each solution (CLX controls on top of Qt, VCL controls on top of Windows).

NOTE *Distributing Qt applications on Windows implies the distribution of the Qt library itself. On the Linux platform, you can generally take the presence of the Qt library for granted, but you still have the interface library to deploy. Also, Linux Borland's CLX is tied to a specific version of Qt (which in Kylix 3 has been specifically patched by Borland), so you'll probably have to distribute it anyway. Distributing the Qt libraries with a professional application (as opposed to an open source project) generally implies paying a license to Trolltech. If you use Delphi or Kylix to build Qt applications, however, Borland has already paid the license to Trolltech for you. You must use at least one CLX class wrapping Qt: If you use the Qt classes exclusively (no CLX at all), you still owe the license to Qt, even when using Delphi or Kylix.*

Technically, huge differences exist behind the scenes between a native Windows application built with VCL and a portable Qt program developed with VisualCLX. Suffice to say that at the low level, Windows uses API function calls and messages to communicate with controls, whereas Qt uses class methods and direct method callbacks and has no internal messages. Technically, the Qt classes offer a high-level object-oriented architecture, but the Windows API is still bound to its C legacy and a message-based system dated 1985 (when Windows was released). VCL offers an object-oriented abstraction on top of a low-level API, whereas VisualCLX remaps an already high-level interface into a more *familiar* class library. (For more on the Qt architecture, see the following sidebar "From Qt to CLX.")

NOTE *Microsoft has reached the point of starting to abandon the traditional low-level Windows API for a native high-level class library, part of the .NET architecture. You can read more about this topic in Part IV of this book.*

If the underlying architectures of the Windows API on one side and Qt on the other side are relevant, the two class libraries built by Borland (VCL and CLX) flatten out most differences, making the code of Delphi and Kylix applications extremely similar. Using VisualCLX on Linux offers Delphi programmers the advantage of having a familiar class library on top of a totally new platform.

From the outside, a button is an object of the `TButton` class for both libraries, and it has more or less the same set of methods, properties, and events. In many cases, you can recompile your existing programs for the new class library in a matter of minutes, if they don't use low-level API calls, platform-dependent features (like ADO or COM), or legacy features (like the BDE).

FROM QT TO CLX

Qt is a C++ class library that includes a complete set of widgets resembling not only Windows core components (buttons, list boxes, and the like) but also most of the common controls (such as TreeView and ListView controls). Because Qt is a C++ library, it cannot be invoked directly from the Delphi language code. The Qt API is accessible instead through a binding layer, defined in the Qt.pas unit.

This binding layer consists of a long list of wrappers of almost every Qt class suffixed with a final *H*. So, for example, the Qt `QPainter` class becomes the `QPainterH` type in the binding layer. The Qt.pas unit also includes a long list of all the public methods of these classes, transformed into standard functions (not class methods) that have as their first parameter the object to which they apply. The only relevant exception to this approach is the class constructors, which are transformed into functions that return the new instance of the class.

Notice that the use of at least one of the classes of the mapping layer is compulsory for the Qt license that comes with Delphi (and Kylix). Qt is free for non-commercial use under X Window (it is called Qt Free Edition), but you must pay a license fee to Trolltech to develop commercial applications. When you buy Delphi, the Qt license has already been paid for you by Borland, but you must use Qt primarily through CLX (even if low-level calls to Qt within a CLX application are allowed). You cannot use the Qt.pas unit directly and avoid including the QForms unit (which is mandatory). Borland enforces this limitation by omitting from the Qt interface the `QFormH` and `QApplicationH` constructors.

In most of this book's Delphi programs, I'll use only CLX objects and methods. It is important to know that if necessary, you can use some extra Qt features directly; or you may have to do low-level calls to bypass CLX bugs. The Qt documentation is not included in the Delphi help, but you can find it on the Trolltech website (`www.trolltech.com`) in HTML and PDF format.

Delphi's Dual Library Support

Delphi has full support for both libraries at design time and at run time. As you begin developing a new application, you can use the File ➢ New Application command to create a new VCL-based program or File ➢ New CLX Application for a new CLX-based program. After you give one of these commands, Delphi's IDE will create a VCL or CLX design-time form and update the Component Palette so that it displays only the visual components compatible with the type of application you've selected (see Figure 5.1 for a comparison). You cannot place a VCL button into a CLX form, and you cannot even mix forms of the libraries within a single executable file. In other words, the user interface of every application must be built using one of the two libraries exclusively, which (aside from the technical implications) makes a lot of sense to me.

If you haven't already done so, I suggest you to try experimenting with the creation of a CLX application, looking at the available controls and trying to use them. You'll find few differences in the use of the components, and if you have been using Delphi for some time, you'll probably be immediately adept with CLX.

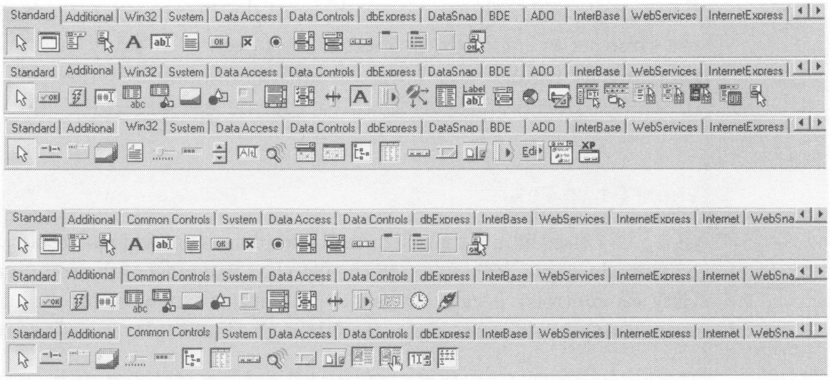

Standard | Additional | Win32 | System | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | WebServices | InternetExpress
Standard | Additional | Common Controls | System | Data Access | Data Controls | dbExpress | InterBase | WebServices | InternetExpress | Internet | WebSna

FIGURE 5.1
A comparison of the first three pages of the Component Palette for a CXL-based application (above) and a VCL-based application (below)

SAME CLASSES, DIFFERENT UNITS

One of the cornerstones of the source-code compatibility between CLX and VCL is the fact that similar classes in the two libraries have the same class name. For example, each library has a class called `TButton` representing a push button; the methods and properties are so similar that this code will work with both libraries:

```
with TButton.Create (Self) do
begin
  SetBounds (20, 20, 80, 35);
  Caption := 'New';
  Parent := Self;
end;
```

The two `TButton` classes can have the same name because they are saved in two different units, called StdCtrls and QStdCtrls. Of course, you cannot have the two components available at design time in the palette, because the Delphi IDE can register only components with unique names. The entire VisualCLX library is defined by units corresponding to the VCL units, but with the letter *Q* as a prefix—so there is a QForms unit, a QDialogs unit, a QGraphics unit, and so on. A few peculiar units, such as QStyle, have no corresponding unit in VCL because they map to features of Qt with no correspondence in the Windows API.

Notice that there are no compile settings or other hidden techniques to distinguish between the two libraries; what matters is the set of units referenced in the code. Remember that these references must be consistent—you cannot mix visual controls of the two libraries in a single form or even in a single program.

DFM AND XFM

As you create a form at design time, it is saved to a form definition file. Traditional VCL applications use the DFM extension, which stands for Delphi form module. CLX applications use the XFM extension, which stands for cross-platform (*X*) form module. A form module is the result of streaming the form and its components: The two libraries share the streaming code, so they produce a similar effect. The format of DFM and XFM files, which can be based on a textual or binary representation, is identical.

So, the reason for having two different extensions doesn't lie in internal compiler tricks or incompatible formats. It is merely an indication to programmers and to the IDE of the type of components you should expect to find within that definition (this indication is *not* included in the file).

If you want to convert a DFM file into an XFM file, you can simply rename the file. However, expect to find some differences in the properties, events, and available components—reopening the form definition for a different library will probably cause quite a few warnings.

TIP *Delphi's IDE chooses the active library by looking at the extension of the form module, ignoring the references in the* `uses` *statements. For this reason, you should change the extension if you plan to use CLX. On Kylix, a different extension is useless, because any form is opened in the IDE as a CLX form regardless of the extension. On Linux, there is only the Qt-based CLX library, which is both the cross-platform and the native library.*

As an example, I've built two identical applications, LibComp and QLibComp, with only a few components and a single event handler. Listing 5.1 presents the textual form definitions for two applications, built using the same steps in the Delphi IDE, after choosing a CLX or VCL application. I've marked differences in bold; as you can see, there are very few, most relating to the form and its font. `OldCreateOrder` is a legacy property used for compatibility with Delphi 3 and older code; standard colors have different names; and CLX saves the scroll bars' ranges.

LISTING 5.1: AN XFM FILE (LEFT) AND AN EQUIVALENT DFM FILE (RIGHT)

```
object Form1: TForm1                 object Form1: TForm1
  Left = 192                           Left = 192
  Top = 107                            Top = 107
  Width = 350                          Width = 350
  Height = 210                         Height = 210
  Caption = 'QLibComp'                 Caption = 'LibComp'
  Color = clBackground                 Color = clBtnFace
  VertScrollBar.Range = 161            Font.Charset = DEFAULT_CHARSET
  HorzScrollBar.Range = 297            Font.Color = clWindowText
                                       Font.Height = -11
                                       Font.Name = 'MS Sans Serif'
                                       Font.Style = []
  TextHeight = 13                      TextHeight = 13
  TextWidth = 6                        OldCreateOrder = False
  PixelsPerInch = 96                   PixelsPerInch = 96
  object Button1: TButton              object Button1: TButton
    Left = 56                            Left = 56
    Top = 64                             Top = 64
    Width = 75                           Width = 75
    Height = 25                          Height = 25
    Caption = 'Add'                      Caption = 'Add'
    TabOrder = 0                         TabOrder = 0
    OnClick = Button1Click               OnClick = Button1Click
  end                                  end
  object Edit1: TEdit                  object Edit1: TEdit
    Left = 40                            Left = 40
    Top = 32                             Top = 32
    Width = 105                       Width = 105
    Height = 21                       Height = 21
    TabOrder = 1                      TabOrder = 1
    Text = 'my name'                  Text = 'my name'
  end                               end
  object ListBox1: TListBox         object ListBox1: TListBox
    Left = 176                        Left = 176
    Top = 32                          Top = 32
    Width = 121                       Width = 121
    Height = 129                      Height = 129
    Rows = 3                          ItemHeight = 13
    Items.Strings = (                 Items.Strings = (
      'marco'                           'marco'
      'john'                            'john'
      'helen')                          'helen')
    TabOrder = 2                      TabOrder = 2
  end                               end
end                               end
```

USES STATEMENTS

If you look at the source code of a VCL or CLX application, the only relevant difference relates to the `uses` statements. The form of the CLX application has the following initial code:

```
unit QLibCompForm;
interface
uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;
```

The form of the VCL program has the traditional `uses` statement:

```
unit LibCompForm;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
```

The code of the class and of the only event handler is absolutely identical. Of course, the compiler directive `{$R *.dfm}` is replaced by `{$R *.xfm}` in the CLX version of a standard program.

DISABLING THE DUAL LIBRARY HELP SUPPORT

When you press the F1 key in the editor to ask for help on a routine, class, or method of the Delphi library, you'll usually get a choice between the VCL and CLX declarations of the same feature. You'll need to make a choice to proceed to the related help page, which can be quite annoying after a while (especially because the two pages are often identical).

If you don't care about CLX and are planning to use only VCL (or vice versa), you can disable this alternative by choosing the Help ➢ Customize command, removing everything with *CLX* in the name from Contents, Index, and Link, and saving the project. Then restart the Delphi IDE, and the Help engine won't bother asking you about CLX any more. Of course, don't forget to add those help files again if you decide to begin using CLX. Similarly, you can reduce the memory occupation and load time of the Delphi IDE by uninstalling all the CLX-related packages.

Choosing a Visual Library

Because two different user interface libraries are available in Delphi, you'll have to choose one for each *visual* application. You must evaluate multiple criteria to come to the proper decision, which isn't always easy.

The first criterion is portability. If running your program on Windows and on Linux, with the same user interface, is a major concern to you, then using CLX will make your life simpler and let you keep a single source code file with limited IFDEFs. The same applies if you consider Linux to be (or think it possibly will become) your key platform. On the other hand, if most of your users are on Windows and you just want to extend your offering with a Linux version, you might want to keep a dual VCL/CLX system. Doing so means you'll probably need two different sets of source code files, or you may have too many IFDEFs.

Another criterion is the native look-and-feel. Is you use CLX on Windows, some controls will behave slightly differently than users expect—at least expert users. For a simple user interface (edits, buttons, grids), this probably won't matter much; but if you have many tree view and list view controls, the differences will be clear. On the other hand, with CLX, you'll be able to let your users choose a look-and-feel that's different from the basic Windows look, and use it consistently across platforms. This means that a Motif fan will be able to choose this style even when forced to use the Windows platform. While this flexibility is common on Linux, you'll seldom use a non-native look-and-feel on Windows.

Using native controls also implies that as soon as you get a new version of the Windows operating system, your application will (probably) adapt to it. This is good for the user, but might cause you a lot of headaches in case of incompatibilities. Differences in the Microsoft common controls library over the last few years have been a major source of frustration for Windows programmers in general, including Delphi programmers.

Another criterion is deployment: If you use CLX, you'll have to ship your Windows and Linux program with the Qt libraries.

I've done a little testing, and the speed of VCL and CLX applications is similar. I've tried creating 1,000 components and showing them on screen, and the speed differences are few; the VCL-based solution offered a 30 percent advantage (for what this limited benchmark is worth). You can try my tests with the LibSpeed and QLibSpeed examples for this chapter.

Another important criterion for deciding to use CLX instead of VCL is a need for Unicode support. CLX has Unicode support in controls by default (even on Win9*x* platforms where it isn't supported by Microsoft). VCL, however, has very little Unicode support even on versions of Windows that provide it, making it difficult to build VCL apps for countries where the local character is managed more easily when it is Unicode based.

RUNNING IT ON LINUX

The real issue of choosing the library resolves to the importance of Linux or Unicode for you and your users. It's important to notice that if you create a CLX application, you'll be able to recompile it unchanged (with the exact source code) with Kylix, producing a native Linux application, unless you've done any Windows API programming at all, in which case conditional compilation will be essential.

As an example, I've recompiled the QLibComp example introduced earlier. Figure 5.2 shows it running; you can also see the Kylix IDE in action on KDE.

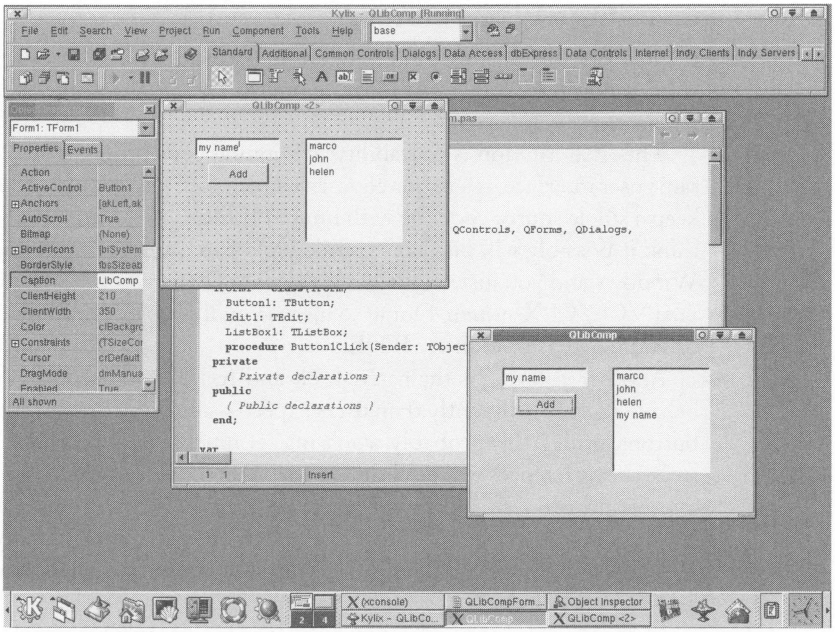


FIGURE 5.2
An application written with CLX can be directly recompiled under Linux with Kylix (displayed in the background).

Kylix - QLibComp [Running]
File Edit Search View Project Run Component Tools Help
base
Standard
Additional
Common Controls
Dialogs
Data Access
dbExpress
Data Controls
Internet
Indy Clients
Indy Servers
QLibComp <2>
m.pas
QLibComp
Add
1 1
Insert
X (xconsole)
QLibCompForm...
Object Inspector
Kylix - QLibCo...
QLibComp
QLibComp <2>
2
4

Form1: TForm1

Properties | Events

Action	
ActiveControl	Button1
⊞Anchors	[akLeft,ak
AutoScroll	True
Bitmap	(None)
⊞BorderIcons	[biSystem
BorderStyle	fbsSizeab
Caption	LibComp
ClientHeight	210
ClientWidth	350
Color	clBackgro
⊞Constraints	(TSizeCor
Cursor	crDefault
DragMode	dmManua
Enabled	True

All shown

my name

marco
john
helen

my name

marco
john
helen
my name

```
QControls, QForms, QDialogs,

    Button1: TButton;
    Edit1: TEdit;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObjec
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

CONDITIONAL COMPILATION FOR LIBRARIES

If you want to keep a single source code file but compile with VCL on Windows and CXL on Linux, you can use platform-specific symbols (such as `$IFDEF LINUX`) to distinguish the two situations in case of conditional compilation. But what if you want to be able to compile a portion of code for both libraries on Windows?

You can either define a symbol of your own and use conditional compilation, or (at times) test for the presence of identifiers that exist only in VCL or CLX:

```
{$IF Declared(QForms)}
   ...CLX-specific code
{$IFEND}
```

Converting Existing Applications

Besides starting with new CLX applications, you might want to convert some of your existing VCL applications to the new class library. You must perform a series of operations without any specific help from the Delphi IDE:

- Rename the DFM file using the XFM extension and update all the `{$R *.DFM}` statements as `{$R *.XFM}`.

- Update all the `uses` statements in your program (in the units and project files) to refer to the CLX units instead of the VCL units. If you miss even a few, you'll bump into trouble when running your application.

TIP *To prevent a CLX application from compiling if it contains references to VCL units, you can move the VCL units to a different directory under* `lib` *and avoid including this folder in your search path. This way, leftover references to VCL units will cause a "Unit not found" error.*

Table 5.1 compares the names of the visual VCL and CLX units, excluding the database portion and some rarely referenced units.

TABLE 5.1: NAMES OF EQUIVALENT VCL AND CLX UNITS

VCL	CLX
ActnList	QActnList
Buttons	QButtons
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	QConsts
Controls	QControls
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdCtrls	QStdCtrls

You might also convert references to Windows and Messages into references to the Qt unit. Some Windows data structures are now also available in the Types unit (see Chapter 3, "The Run-Time Library," for details), so you might have to add it to your CLX programs. Notice, however, that the QTypes unit is not the CLX version of VCL's Types unit; these two units are totally unrelated.

WARNING *Watch out for your* `uses` *statements! If you compile a project that includes a CLX form, but you fail to update the project source code, leaving a reference to the VCL Forms unit there, your program will run but stop immediately. The reason is that no VCL form was created, so the program terminated right away. In other cases, trying to create a CLX form within a VCL application will cause run-time errors. Finally, the Delphi IDE might inappropriately add references to* `uses` *statements of the wrong library; in this case you end up with a single* `uses` *statement that refers to the same unit for both libraries, but only the second of the two will be effective. This situation rarely prevents the program from compiling, but you won't be able to run it.*

As a helper in converting some of my own programs, I've written a simple unit-replacement tool called VclToClx; it's available in the Tools section of the source code that accompanies the book. You can find more information about this program in Appendix A, "Extra Delphi Tools by the Author."

TControl and Derived Classes

In Chapter 4, I discussed the base classes of the Delphi library, focusing particularly on the `TComponent` class. One of the most important subclasses of `TComponent` is `TControl`, which corresponds to visual components. This base class is available in both CLX and VCL and defines general concepts, such as the position and the size of the control, the parent control hosting it, and more. For an actual implementation, though, you have to refer to its two subclasses. In VCL these are `TWinControl` and `TGraphicControl`; in CLX they are `TWidgetControl` and `TGraphicControl`. Here are their key features:

Window-Based Controls (Also Called Windowed Controls) Visual components based on an operating-system window. A `TWinControl` in VCL has a window handle, which is a number referring to an internal Windows structure. A `TWidgetControl` in CLX has a Qt handle, which is a reference to the internal Qt object. From a user perspective, windowed controls can receive the input focus, and some of them can contain other controls. This is the biggest group of components in the Delphi library. We can further divide windowed controls into two groups: wrappers of native controls of Windows or Qt; and custom controls, which generally inherit from `TCustomControl`.

Graphical Controls (Also Called Nonwindowed Controls) Visual components not based on an operating-system window. Therefore, these components have no handle, cannot receive the focus, and cannot contain other controls. They inherit from `TGraphicControl` and are painted by their parent control, which sends them mouse-related and other events. Examples of nonwindowed controls are the Label and SpeedButton components. There are just a few controls in this group, which were critical to minimizing the use of system resources in the early days of Delphi (on 16-bit Windows). Using graphical controls to save Windows resources is still useful on Win9*x*/Me, which has pushed the system limits higher but hasn't fully gotten rid of them (unlike Windows NT/2000).

Parent and Controls

The `Parent` property of a control indicates which other control is responsible for displaying it. When you drop a component into a form in the Form Designer, the form will become both parent and owner of the new control. But if you drop the component inside a Panel, ScrollBox, or any other *container* component, this will become its parent, whereas the form will still be the owner of the control.

When you create the control at run time, you'll need to set the owner (using the `Create` constructor's parameter); but you must also set the `Parent` property, or the control won't be visible.

Like the `Owner` property, the `Parent` property has an inverse. The `Controls` array lists all the controls parented by the current one, numbered from `0` to `ControlCount - 1`. You can scan this property to operate on all the controls hosted by another control, eventually using a recursive method that operates on the controls parented by each subcontrol.

Properties Related to Control Size and Position

Some of the properties introduced by `TControl` and common to all controls are those related to size and position. The position of a control is determined by its `Left` and `Top` properties, and its size is specified by the `Height` and `Width` properties. Technically, all components have a position, because when you reopen an existing form at design time, you want to be able to see the icons for the nonvisual components in exactly the position where you've placed them. This position is visible in the form file.

> *TIP* *As you change any of the positional or size properties, you end up calling the single* `SetBounds` *method. So, any time you need to change two or more of these properties at once, calling* `SetBounds` *directly will speed up the program. Another method,* `BoundsRect`*, returns the rectangle bounding the control and corresponds to accessing the properties* `Left`*,* `Top`*,* `Height`*, and* `Width`*.*

An important feature of the position of a component is that, like any other coordinate, it always relates to the client area of its parent component (indicated by its `Parent` property). For a form, the client area is the surface included within its borders and caption (excluding the borders themselves). It would have been messy to work in screen coordinates, although some ready-to-use methods convert the coordinates between the form and the screen and vice versa.

Note, however, that the coordinates of a control are always relative to the parent control, such as a form or another *container* component. If you place a panel in a form and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In this case, the parent component of the button is the panel.

Activation and Visibility Properties

You can use two basic properties to let the user activate or hide a component. The simpler is the `Enabled` property. When a component is disabled (when `Enabled` is set to False), usually some visual hint indicates this state to the user. At design time, the "disabled" property does not always have an effect; but at run time, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding `Hide` method or by setting its `Visible` property to False. Be aware, however, that reading the status of the `Visible` property does not tell you whether the control is actually visible. If the container of a control is hidden, even if the control is set to `Visible`, you cannot see it. For this reason, you can read the value of the run-time, read-only `Showing` property to determine whether the control is really visible to the user; that is, if it is visible, its parent control is also visible, the parent control of the parent control is also visible, and so on.

Fonts

The `Color` and `Font` properties are often used to customize the user interface of a component. Several properties are related to the color. The `Color` property itself usually refers to the background color of the component. There is also a `Color` property for fonts and many other graphical elements. Many components also have `ParentColor` and `ParentFont` properties, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the `Font` property of the form itself.

When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Delphi allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users' computers.

If your program uses a font that your user doesn't have, Windows will select some other font to use in its place. A program's carefully formatted output can be ruined by the font substitution. For this reason, you should rely on standard Windows fonts (such as MS Sans Serif, System, Arial, Times New Roman, and so on).

Colors

There are various ways to set the value of a color. The type of this property is `TColor`, which isn't a class type but just an integer type. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include clBlue, clSilver, clWhite, clGreen, clRed, and many others (including Delphi 6's clMoneyGreen, clSkyBlue, clCream, and clMedGray). As a better alternative, you can use one of the colors used by the system to denote the status of given elements. These sets of colors are different in VCL and CLX.

VCL includes predefined Windows colors such as the background of a window (clWindow), the color of the text of a highlighted menu (clHighlightText), the active caption (clActiveCaption), and the ubiquitous button face color (clBtnFace). CLX includes a different and incompatible set of system colors, including clBackground, which is the standard color of a form; clBase, used by edit boxes and other visual controls; clActiveForeground, the foreground color for active controls; and clDisabledBase, the background color for disabled text controls. All the color constants mentioned here are listed in VCL and CLX Help files under the "TColor type" topic.

Another option is to specify a `TColor` as a number (a 4-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low three bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value $00FF0000 corresponds to a pure blue color, the value $0000FF00 to green, the value $000000FF to red, the value $00000000 to black, and the value $00FFFFFF to white. By specifying intermediate values, you can obtain any of 16 million possible colors.

Instead of specifying these hexadecimal values directly, you should use the Windows `RGB` function, which has three parameters, all ranging from 0 to 255. The first indicates the amount of red, the second the amount of green, and the last the amount of blue. Using the `RGB` function makes programs generally more readable than using a single hexadecimal constant. `RGB` is *almost* a Windows API function; it is defined by the Windows-related units and not by Delphi units, but a similar function does not exist in the Windows API. C includes a macro that has the same name and effect, which is a welcome addition to the Windows unit. `RGB` is not available on CLX, so I've written my own version:

```
function RGB (red, green, blue: Byte): Cardinal;
begin
```

```
    Result := blue + green * 256 + red * 256 * 256;
  end;
```

The highest-order byte of the `TColor` type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.)

Regarding palettes and color matching, note that Windows sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, Windows uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors. In 16-color (VGA) adapters and at higher resolutions, you often end up seeing strange patterns of pixels of different colors rather than the color you had in mind.

The *TWinControl* Class (VCL)

In Windows, most elements of the user interface are windows. From a user standpoint, a window is a portion of the screen surrounded by a border, having a caption and usually a system menu. But technically speaking, a window is an entry in an internal system table, often corresponding to an element visible on the screen that has some associated code. Most of these windows have the role of controls; others are temporarily created by the system (for example, to show a pull-down menu). Still other windows are created by the application but remain hidden from the user and are used only as a way to receive a message (for example, nonblocking sockets use windows to communicate with the system).

The common denominator of all windows is that they are known by the Windows system and refer to a function for their behavior; each time something happens in the system, a notification message is sent to the proper window, which responds by executing some code. Each window of the system has an associated function (generally called its *window procedure*), which handles the various messages the window is interested in.

In Delphi, any `TWinControl` class can override the `WndProc` method or define a new value for the `WindowProc` property. Interesting Windows messages, however, can be better tracked by providing specific message handlers. Even better, VCL converts these lower-level messages into events. In short, Delphi allows you to work at a high level, making application development easier, but still allows you to go low-level when required.

Notice also that creating an instance of a `TWinControl`-based class doesn't automatically create its corresponding Window handle. Delphi uses a lazy initialization technique, so that the low-level control is created only when required—generally as soon as a method accesses the `Handle` property. The `get` method for this property calls `HandleNeeded` the first time, which eventually calls `CreateHandle`... and so on, eventually reaching `CreateWnd`, `CreateParams`, and `CreateWindowHandle` (the sequence is complex, and you don't need to know it in detail). At the opposite end, you can keep an existing (perhaps invisible) control in memory but destroy its window handle, to save system resources.

The *TWidgetControl* Class (CLX)

In CLX, every `TWidgetControl` has an internal Qt object, referenced using the `Handle` property. This property has the same name as the corresponding Windows property, but it is totally different behind the scenes.

A `TWidgetControl` object generally owns the corresponding Qt/C++ object. The CLX class uses delayed construction (the internal object is not created until one of its methods is required) implemented in `InitWidget` and other methods. The CLX class also frees the internal object when it is destroyed. However, it is also possible to create a widget around an existing Qt object: In this case, the CLX object won't own the Qt object and won't destroy it. This behavior is indicated in the `OwnHandle` property.

To be more precise, each VisualCLX component has two associated C++ objects: the Qt `Handle` and the Qt `Hook`, which is the object receiving the system events. With the current Qt design, the Qt `Hook` must be a C++ object, which acts as an intermediary to the event handlers of the Delphi language control. The `HookEvents` method associates the hook object to the CLX control.

Unlike Windows, Qt defines two different types of events:

- *Events* are the translation of user input or system events (such as keystroke, mouse move, and paint).
- *Signals* are internal component events (corresponding to VCL internal or abstract operations, such as `OnClick` and `OnChange`).

The events of a CLX component, however, merge events and signals. Generic Delphi CLX control events include `OnMouseDown`, `OnMouseMove`, `OnKeyDown`, `OnChange`, `OnPaint`, and many others, exactly as in the VCL (which fires most events as a response to Windows messages).

NOTE *Expert programmers may notice that CLX includes a seldom-used* `EventHandler` *method, which corresponds more or less to the* `WndProc` *method of VCL* `TWinControl`*.*

Opening the Component Toolbox

So, you want to write a Delphi application. You open a new Delphi project and find yourself faced with a large number of components. The problem is that for every operation, there are multiple alternatives. For example, you can show a list of values using a list box, a combo box, a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which should you use? That's difficult to say. There are many considerations, depending on what you want your application to do. For this reason, I've provided a highly condensed summary of alternative options for a few common tasks.

NOTE *For some of the controls described in the following sections, Delphi also includes a data-aware version, usually indicated by the DB prefix. As you'll see in Chapter 13, "Delphi's Database Architecture," the DB version of a control typically serves a role similar to that of its "standard" equivalent; but the properties and the ways you use it are often quite different. For example, in an Edit control you use the* `Text` *property, whereas in a DBEdit component you access the* `Value` *of the related field object.*

The Text Input Components

Although a form or component can handle keyboard input directly using the `OnKeyPress` event, this isn't a common operation. Windows provides ready-to-use controls you can use to get string input and even build a simple text editor. Delphi has several slightly different components in this area.

THE EDIT COMPONENT

The Edit component allows the user to enter a single line of text. You can also display a single line of text with a Label or a StaticText control, but these components are generally used only for fixed text or program-generated output, not for input. In CLX, there is also a native LCD digit control you can use to display numbers.

The Edit component uses the `Text` property, whereas many other controls use the `Caption` property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the `OnKeyPress` event of the edit box. For example, you can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's not, you change the value of the key to the null character (`#0`), so that it won't be processed by the edit control and will produce a warning beep:

```
procedure TForm1.Edit1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

NOTE *A minor difference of CLX is that the Edit control has no built-in Undo mechanism. Another is that the* `PasswordChar` *property is replaced by the* `EchoMode` *property. You don't determine the character to display, but whether to echo the entered text or display an asterisk instead.*

THE LABELEDEDIT CONTROL

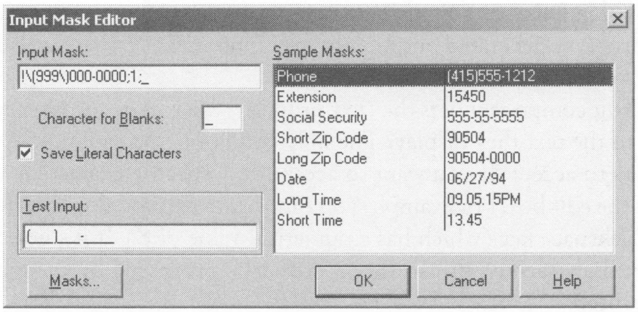
Delphi 6 added a nice control called LabeledEdit, which is an Edit control with a label attached to it. The Label appears as a property of the compound control, which inherits from `TCustomEdit`.

This component is very handy, because it allows you to reduce the number of components on your forms, move them around more easily, and have a more consistent layout for all of the labels of an entire form or application. The `EditLabel` property is connected with the subcomponent, which has the usual properties and events. Two more properties, `LabelPosition` and `LabelSpacing`, allow you to configure the relative positions of the two controls.

NOTE *This component has been added to the ExtCtrls unit to demonstrate the use of subcomponents in the Object Inspector. I'll discuss the development of these components in Chapter 9, "Writing Delphi Components." Notice also that this component is not available on CLX.*

THE MASKEDIT COMPONENT

To customize the input of an edit box further, you can use the MaskEdit component. It has an `EditMask` property, which is a string indicating for each character whether it should be uppercase, lowercase, or a number, and other similar conditions. You can see the editor for the `EditMask` property here:



Input Mask: !\(999\)000-0000;1;_

Character for Blanks: _

☑ Save Literal Characters

Test Input: (___)___-____

Sample Masks:

Phone	(415)555-1212
Extension	15450
Social Security	555-55-5555
Short Zip Code	90504
Long Zip Code	90504-0000
Date	06/27/94
Long Time	09.05.15PM
Short Time	13.45

The Input Mask Editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to display the parentheses around the area code of a phone number only as an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask Editor correspond to the last two fields of the mask (separated by semicolons).

TIP *Clicking the Masks button in the Input Mask Editor lets you choose predefined input masks for different countries.*

THE MEMO AND RICHEDIT COMPONENTS

The controls discussed so far allow a single line of input. The Memo component, by contrast, can host several lines of text but (on the Win95/98 platforms) still retains the 16-bit Windows text limit (32 KB) and allows only a single font for the entire text. You can work on the text of the memo line by line (using the `Lines` string list) or access the entire text at once (using the `Text` property).

If you want to host a large amount of text or change fonts and paragraph alignments, in VCL you should use the RichEdit control, a Win32 common control based on the RTF document format. You can find an example of a complete editor based on the RichEdit component among the sample programs that ship with Delphi. (The example is named RichEdit, too.)

WARNING *The RichEdit control is one of the few commonly used Delphi controls not available in CXL and in Kylix. The latest version of Qt has a similar native control, so these controls may be supported by future versions of CLX.*

The RichEdit component has a `DefAttributes` property indicating the default styles and a `SelAttributes` property indicating the style of the current selection. These two properties are not of the `TFont` type, but they are compatible with fonts, so you can use the `Assign` method to copy the value, as in the following code fragment:

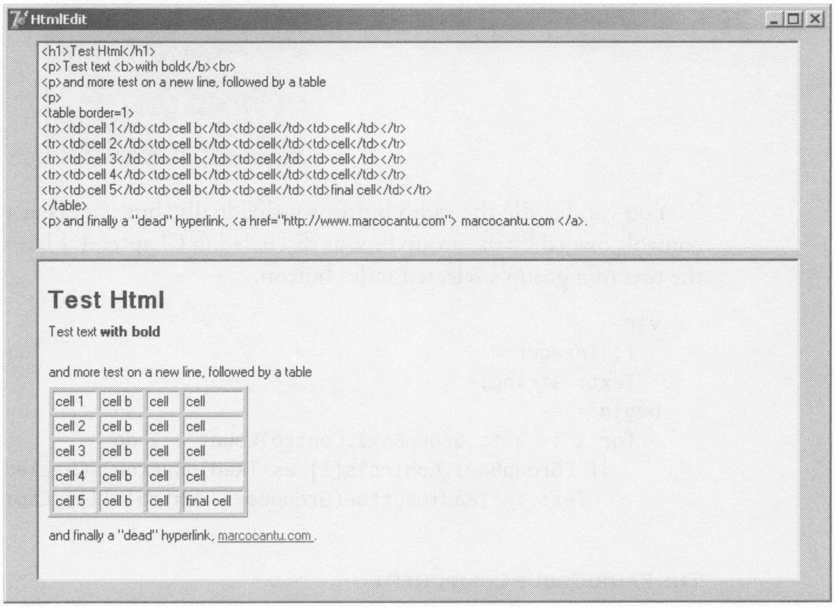
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign (RichEdit1.DefAttributes);
    if FontDialog1.Execute then
```

```
      RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  end;
end;
```

THE TEXTVIEWER CLX CONTROL

CLX and Qt lack a RichEdit control, but on the other hand they provide a full-blown HTML viewer, which is powerful for displaying formatted text but not for typing it. This HTML viewer is embedded in two controls: the single-page TextViewer control and the TextBrowser control with active links.

As a simple demo, I've added a memo and a text viewer to a CLX form and connected them so that everything you type on the memo is immediately displayed in the viewer. I've called the example HtmlEdit not because this is a real HTML editor, but because this is the simplest way I know to build an HTML preview inside a program. The program's form is shown at run time in Figure 5.3.



```
<h1>Test Html</h1>
<p>Test text <b>with bold</b><br>
<p>and more test on a new line, followed by a table
<p>
<table border=1>
<tr><td>cell 1</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 2</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 3</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 4</td><td>cell b</td><td>cell</td><td>cell</td></tr>
<tr><td>cell 5</td><td>cell b</td><td>cell</td><td>final cell</td></tr>
</table>
<p>and finally a "dead" hyperlink, <a href="http://www.marcocantu.com"> marcocantu.com </a>.
```

Test Html

Test text **with bold**

and more test on a new line, followed by a table

cell 1	cell b	cell	cell
cell 2	cell b	cell	cell
cell 3	cell b	cell	cell
cell 4	cell b	cell	cell
cell 5	cell b	cell	final cell

and finally a "dead" hyperlink, marcocantu.com .

FIGURE 5.3
The HtmlEdit example at run time: When you add new HTML text to the memo, you get an immediate preview.

TIP I originally built this example with Kylix on Linux. To port it to Windows and Delphi, all I had to do was copy the files and recompile.

Selecting Options

Two standard Windows controls allow the user to choose different options. Two other controls let you group sets of options.

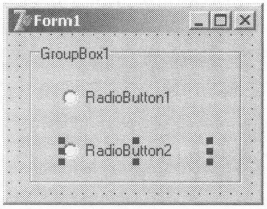
THE CHECKBOX AND RADIOBUTTON COMPONENTS

The first standard option-selecting control is the *check box*, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the `AllowGrayed` property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks the check box.

The second type of control is the *radio button*, which corresponds to an exclusive selection. Two radio buttons on the same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected (as programmer, you are responsible for selecting one of the radio buttons at design time).

THE GROUPBOX COMPONENTS

To host several groups of radio buttons, you can use a GroupBox control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the GroupBox component on a form and then add the radio buttons to the group box, as in the following example:



You can handle the radio buttons individually, but it's easier to navigate through the array of controls owned by the group box, as discussed in Chapter 4. Here is a small code excerpt used to get the text of a group's selected radio button:

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := TRadioButton(GroupBox1.Controls[I]).Caption;
```

THE RADIOGROUP COMPONENT

Delphi has a similar component that can be used specifically for radio buttons: the RadioGroup component. A RadioGroup is a group box with some radio buttons inside it. The difference is that these internal radio buttons are managed automatically by the container control. Using a radio group is generally easier than using a group box, because the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Another advantage is that the RadioGroup component can automatically align its radio buttons in one or more columns (as indicated by the `Columns` property), and you can easily add new choices at run time by adding strings to the `Items` string list. By contrast, adding new radio buttons to a group box is quite complex.

Lists

When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six, to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that display lists of items and allow the user to select one of them.

THE LISTBOX COMPONENT

The selection of an item in a list box uses the `Items` and `ItemIndex` properties as in the earlier code shown for the RadioGroup control. If you need access to the text of selected list box items often, you can write a small wrapper function like this:

```
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := '';
end;
```

Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the `MultiSelect` property. There are two kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case, a user selects multiple items simply by clicking them; in the second case, the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items, respectively. The two alternatives are determined by the status of the `ExtendedSelect` property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the `SelCount` property, and it can determine which items are selected by examining the `Selected` array. This array of Boolean values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the `Selected` array as follows:

```
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```

Differently from VCL, in CLX you can configure a ListBox to use a fixed number of columns and rows, using the `Columns`, `Row`, `ColumnLayout`, and `RowLayout` properties. Of these, the VCL ListBox has only the `Columns` property.

THE COMBOBOX COMPONENT

List boxes take up a lot of screen space, and they offer a fixed selection—that is, a user can choose only among the items in the list box and cannot enter any choice the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control, which combines an edit box and a drop-down list. The behavior of a ComboBox component changes a lot depending on the value of its `Style` property:

- The csDropDown style defines a typical combo box, which allows direct editing and displays a list box on request.
- The csDropDownList style defines a combo box that does not allow editing (but uses the keystrokes to select an item).
- The csSimple style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a ComboBox is easier than doing the same operation for a list box, because you can simply use the `Text` property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the Enter key. The following method first tests whether the user has pressed that key, by looking for the character with the numeric (ASCII) value of 13. It then tests to make sure the text of the combo box is not empty and is not already in the list (if its position in the list is less than zero). Here is the code:

```
procedure TForm1.ComboBox1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // if the user presses the Enter key
  if Key = Chr (13) then
    with Sender as TComboBox do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

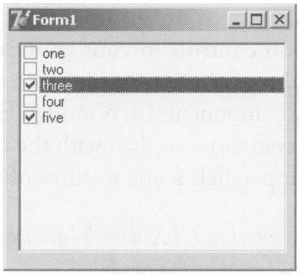
TIP *In CLX, the combo box can automatically add the text typed into the edit box to the drop-down list when the user presses the Enter key. Also, some events fire at different times than in VCL.*

Since version 6, Delphi includes two new events for the combo box. The `OnCloseUp` event corresponds to the closing of the drop-down list and complements the preexisting `OnDropDown` event. The `OnSelect` event fires only when the user selects something in the drop-down list, as opposed to typing in the edit portion.

Another nice feature is the `AutoComplete` property. When it is set, the ComboBox component (and the ListBox, as well) automatically locates the string nearest to the one the user is entering, suggesting the final part of the text. The core of this feature, also available in CLX, is implemented in the `TCustomListBox.KeyPress` method.

THE CHECKLISTBOX COMPONENT

Another extension of the list box control is represented by the CheckListBox component, a list box with each item preceded by a check box:



☐ one
☐ two
☑ three
☐ four
☑ five

A user can select a single item in the list, but can also click the check boxes to toggle their status. This makes the CheckListBox a very good component for multiple selections or for highlighting the status of a series of independent items (as in a series of check boxes).

To check the current status of each item, you can use the `Checked` and `State` array properties (use the latter if the check boxes can be grayed). Delphi 5 introduced the `ItemEnabled` array property, which you can use to enable or disable each item of the list. You'll use the CheckListBox in the DragList example, later in this chapter.

TIP *Most of the list-based controls share a common and important feature: Each item in the list has an associated 32-bit value, usually indicated by the* `TObject` *type. This value can be used as a tag for each list item, and it's very useful for storing additional information along with each item. This approach is connected to a specific feature of the native Windows list box control, which offers four bytes of extra storage for each list box item. You'll use this feature in the ODList example later in this chapter.*

THE EXTENDED COMBO BOXES: COMBOBOXEX AND COLORBOX

The ComboBoxEx (where *ex* stands for extended) is the wrapper of a new Win32 common control that extends the traditional combo box by allowing images to appear next to the items in the list. You attach an image list to the combo box, and then select an image index for each item to display. The effect of this change is that the simple `Items` string list is replaced by a more complex collection, the `ItemsEx` property. I'll use the ComboBoxEx control in the RefList2 example in Chapter 7, "Working with Forms."

TIP *In Delphi 7, the ComboBoxEx component has the new* `AutoCompleteOptions` *property, enabling the combo box to respond to user keystrokes.*

The ColorBox control is a version of the combo box specifically aimed at selecting colors. You can use its `Style` property to choose which groups of colors you want to see in the list (standard color, extended colors, system colors, and so on).

THE LISTVIEW AND TREEVIEW COMPONENTS

If you want an even more sophisticated list, you can use the ListView common control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described in the section "ListView and TreeView Controls" later in this chapter. Other alternatives for listing values are the TreeView common control, which shows items in a hierarchical output, and the StringGrid control, which shows multiple elements for each line. For an actual example of the use of this component, refer to the free online chapter "Graphics in Delphi" discussed in Appendix C.

If you use the common controls in your application, users will already know how to interact with them, and they will regard the user interface of your program as up to date. TreeView and ListView are the two key components of Windows Explorer, and you can assume that many users will be familiar with them—even more so than with the traditional Windows controls. CLX adds also an IconView control, which parallels some features of the VCL ListView.

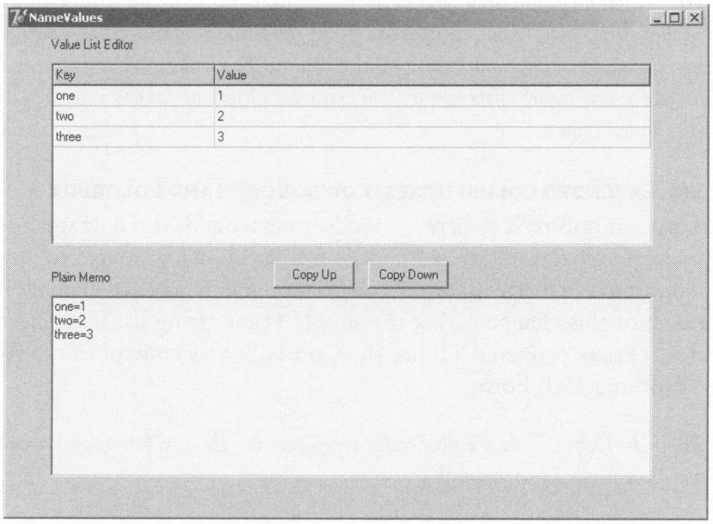
WARNING *The ListView control in CLX doesn't have the small/large icon styles of its Windows counterpart, but a companion control, IconView, provides this capability.*

THE VALUELISTEDITOR COMPONENT

Delphi applications often use the name/value structure natively offered by string lists, which I discussed in Chapter 4. Delphi 6 introduced a version of the StringGrid (technically a `TCustomDrawGrid` descendant class) component specifically geared toward this type of string lists. The ValueListEditor has two columns in which you can display and let the user edit the contents of a string list with name/value pairs, as you can see in Figure 5.4. This string list is indicated in the `Strings` property of the control.

FIGURE 5.4

The NameValues example uses the ValueListEditor component, which shows the name/value or key/value pairs of a string list, also visible in a plain memo.



Key	Value
one	1
two	2
three	3

```
one=1
two=2
three=3
```

The power of this control lies in the fact that you can customize the editing options for each position of the grid or for each key value, using the run-time-only `ItemProps` array property. For each item, you can indicate:

- Whether it is read-only
- The maximum number of characters of the string

- An edit mask (eventually requested in the `OnGetEditMask` event)
- The items in a drop-down pick list (eventually requested in the `OnGetPickList` event), as demonstrated by the first item of the example.
- The display of a button that will show an editing dialog box (in the `OnEditButtonClick` event, which the example handles with a message box)

Needless to say, this behavior resembles what is available generally for string grids and the DBGrid control, and also the behavior of the Object Inspector.

The `ItemProps` property must be set up at run time, by creating an object of the `TItemProp` class and assigning it to an index or a key of the string list. To have a default editor for each line, you can assign the same item property object multiple times. In the example, this shared editor sets an edit mask for up to three numbers:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  SharedItemProp := TItemProp.Create (ValueListEditor1);
  SharedItemProp.EditMask := '999;0; ';
  SharedItemProp.EditStyle := esEllipsis;

  FirstItemProp := TItemProp.Create (ValueListEditor1);
  for I := 1 to 10 do
    FirstItemProp.PickList.Add(IntToStr (I));

  Memo1.Lines := ValueListEditor1.Strings;
  ValueListEditor1.ItemProps [0] := FirstItemProp;
  for I := 1 to ValueListEditor1.Strings.Count - 1 do
    ValueListEditor1.ItemProps [I] := SharedItemProp;
end;
```

You must repeat similar code in case the number of lines changes—for example, by adding new elements in the memo and copying them to the value list.

```
procedure TForm1.ValueListEditor1StringsChange(Sender: TObject);
var
  I: Integer;
begin
  ValueListEditor1.ItemProps [0] := FirstItemProp;
  for I := 1 to ValueListEditor1.Strings.Count - 1 do
    if not Assigned (ValueListEditor1.ItemProps [I]) then
      ValueListEditor1.ItemProps [I] := SharedItemProp;
end;
```

NOTE *Reassigning the same editor twice causes trouble, so I've assigned the editor only to the lines that don't already have one.*

Another property, `KeyOptions`, allows you to let the user edit the keys (the names), add new entries, delete existing entries, and allow for duplicated names in the first portion of the string. Oddly enough, you cannot add new keys unless you also activate the edit options, which makes it hard to let the user add extra entries while preserving the names of the basic entries.

Ranges

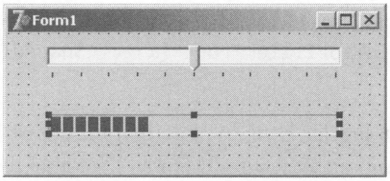
Finally, you can use a few components to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

THE SCROLLBAR COMPONENT

The stand-alone ScrollBar control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border, a feature governed by a specific Windows style of the form window. By *resembling*, I mean that it is not technically a separate window of the ScrollBar component type. These "fake" scroll bars are usually controlled in Delphi using two specific properties of the form and the other components hosting them: `VertScrollBar` and `HorzScrollBar`.

THE TRACKBAR AND PROGRESSBAR COMPONENTS

Direct use of the ScrollBar component is quite rare, especially with the TrackBar component introduced with Windows 95, which is used to let a user select a value in a range. Among Win32 common controls is the companion ProgressBar control, which allows the program to output a value in a range, showing the progress of a lengthy operation. These two components are visible here:



THE UPDOWN COMPONENT

Another related control is the UpDown component, which is usually connected to an edit box so that the user can either type a number in it or increase and decrease the number using the two small arrow buttons. To connect the two controls, you set the `Associate` property of the UpDown component. Nothing prevents you from using the UpDown component as a stand-alone control, displaying the current value in a label or in some other way.

NOTE *CLX has no UpDown control, but it offers a SpinEdit that bundles an Edit with the UpDown in a single control.*

THE PAGESCROLLER COMPONENT

The Win32 PageScroller control is a container allowing you to scroll the internal control. For example, if you place a toolbar in the page scroller and the toolbar is larger than the available area, the PageScroller will display two small arrows on the side. Clicking these arrows will scroll the internal area. This component can be used as a scroll bar, but it also partially replaces the ScrollBox control.

THE SCROLLBOX COMPONENT

The ScrollBox control represents a region of a form that can scroll independently from the rest of the surface. For this reason, the ScrollBox has two scroll bars used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many Windows applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scroll bars of the form, you might allow the user to move the toolbar or status bar out of view (a very odd situation).

Commands

The final category of components is not as clear-cut as the previous ones, and relates to commands. The basic component of this group is the `TButton` (or *push button*, in Windows jargon). More than stand-alone buttons, Delphi programmers use buttons (or `TToolButton` objects) within toolbars (in the early ages of Delphi, they used speed buttons within panels). Beside buttons and similar controls, the other key technique for issuing commands is the use of menu items, part of the *pull-down* menus attached to forms' main menus or local *pop-up* menus activated with the right mouse button.

Menu- or toolbar-related commands fall into different categories depending on their purpose and the feedback their interface provides to the user:

Commands Menu items or buttons used to execute an action.

State-setters Menu items or buttons used to toggle an option on and off, to change the state of a particular element. The menu items of these commands usually have a check mark to their left to indicate that they are active (you can automatically obtain this behavior using the `AutoCheck` property). Buttons are generally painted in a *pressed down* state to indicate the same status (the ToolButton control has a `Down` property).

Radio Items Menus items that display a bullet and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, set the `RadioItem` property to True and set the `GroupIndex` property for the alternative menu items to the same value. In a similar fashion, you can have groups of toolbar buttons that are mutually exclusive.

Dialog Openers Items that cause a dialog box to appear. They are usually indicated by an ellipsis (...) after the text.

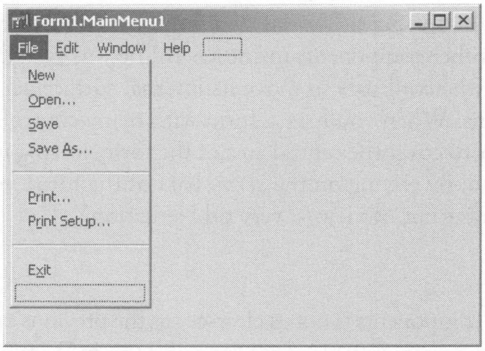
COMMANDS AND ACTIONS

As you'll see in Chapter 6, modern Delphi applications tend to use the ActionList component or its ActionManager extension to handle menu and toolbar commands. In short, you define a series of action objects and associate each of them to a toolbar button and/or a menu item. You can define the command execution in a single place but also update the user interface simply by targeting the action; the related visual control will automatically reflect the status of the action object.

THE MENU DESIGNER

If you just need to show a simple menu in your application, you can place a MainMenu or PopupMenu component on a form and double-click on it to fire up the Menu Designer, shown in Figure 5.5. You add new menu items and provide them with a `Caption` property, using a hyphen (-) to separate caption menu items.

FIGURE 5.5
Delphi's Menu Designer in action



Delphi creates new components for each menu item you add. To name each component, Delphi uses the caption you enter and appends a number (so that *Open* becomes `Open1`). After removing spaces and other special characters from the caption, if nothing is left Delphi adds the letter *N* to the name. Finally it appends the number. Thus menu item separators are called `N1`, `N2`, and so on. Knowing what Delphi tends to do by default, you should think of editing the name first, which is necessary if you want to end up with a sensible component naming scheme.

WARNING *Do not use the* `Break` *property, which is used to lay out a pull-down menu on multiple columns. The mbMenuBarBreak value indicates that this item will be displayed on a second or subsequent line; the mbMenuBreak value means this item will be added to a second or subsequent column of the pull-down menu.*

To obtain a more modern-looking menu, you can add an image list control to the program, hosting a series of bitmaps, and connect the image list to the menu using its `Images` property. You can then set an image for each menu item by setting the proper value of its `ImageIndex` property. The definition of images for menus is quite flexible—you can associate an image list with any specific pull-down menu (and even a specific menu item) using the `SubMenuImages` property. Having a specific smaller image list for each pull-down menu instead of a single large image list for the entire menu allows for more run-time customization of an application.

TIP *Creating menu items at run time is so common that Delphi provides some ready-to-use functions in the Menus unit. The names of these global functions are self-explanatory:* `NewMenu`, `NewPopupMenu`, `NewSubMenu`, `NewItem`, *and* `NewLine`.

POP-UP MENUS AND THE *ONCONTEXTPOPUP* EVENT

The PopupMenu component is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property. However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, which is a label in this code fragment:

```
procedure TForm1.Label3MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
  begin
    ScreenPoint := Label3.ClientToScreen (Point (X, Y));
    PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
  end;
end;
```

An alternative approach is to use the `OnContextMenu` event. This event, introduced in Delphi 5, fires when a user right-clicks a component—exactly what I traced previously with the test `if Button = mbRight`. The advantage is that the same event is also fired in response to a Shift+F10 key combination, as well as the shortcut-menu key of some keyboards. You can use this event to fire a pop-up menu with little code:

```
procedure TFormPopup.Label1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now), 0, False, True, nil, 0, ''));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This result is not particularly useful, but it illustrates that if you need to display a plain pop-up menu, you can easily use the `PopupMenu` property of the control in question or one of its parent controls. Handling the `OnContextMenu` event makes sense only when you want to do some extra processing.

The `Handled` parameter is preinitialized to False, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set `Handled` to True and the system will stop processing the message. You'll rarely set `Handled` to True, because you'll generally handle the `OnContextPopup` to dynamically create or customize the pop-up menu, but then you can let the default handler show the menu.

The handler of an `OnContextPopup` event isn't limited to displaying a pop-up menu. It can perform any other operation, such as directly display a dialog box. Here is an example of a right-click operation used to change the color of the control:

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
  ColorDialog1.Color := Label2.Color;
  if ColorDialog1.Execute then
    Label2.Color := ColorDialog1.Color;
  Handled := True;
end;
```

All the code snippets from this section are available in the simple CustPop example for VCL and QCustPop for CLX.

Control-Related Techniques

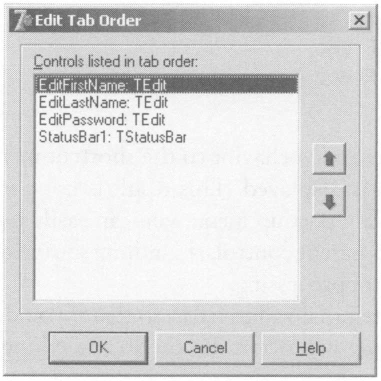
After this general overview of the most commonly used Delphi controls, I'll devote some space to discussing generic core techniques not related to a specific component. I'll cover the input focus, control anchors, the use of the splitter component, and the display of fly-by hints. Of course, these topics don't include everything you can do with visual controls, but they provide a starting point for exploration to get you up and running with some of the most common techniques.

Handling the Input Focus

Using the `TabStop` and `TabOrder` properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key. Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, shown in Figure 5.6.

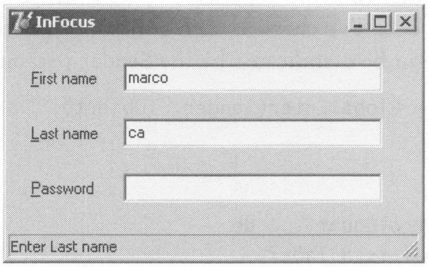
FIGURE 5.6

The Edit Tab Order dialog box



EditFirstName: TEdit
EditLastName: TEdit
EditPassword: TEdit
StatusBar1: TStatusBar

Besides these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the InFocus example, which creates a typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in Figure 5.7. At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.



First name: marco

Last name: ca

Password:

Enter Last name

FIGURE 5.7 The InFocus example at run time

For the output of the status information, I've used the StatusBar component, with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the `&` character in the labels, indicating a shortcut key, and the connection of these labels with corresponding edit boxes (using the `FocusControl` property):

```
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
  object Label1: TLabel
    Caption = '&First name'
    FocusControl = EditFirstName
  end
  object EditFirstName: TEdit
    OnEnter = GlobalEnter
    OnExit = EditFirstNameExit
  end
  object Label2: TLabel
    Caption = '&Last name'
    FocusControl = EditLastName
  end
  object EditLastName: TEdit
    OnEnter = GlobalEnter
  end
  object Label3: TLabel
    Caption = '&Password'
    FocusControl = EditPassword
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    OnEnter = GlobalEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```

The program is simple and performs only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' `OnEnter` event, using a single generic event handler to avoid repetitive code. In the example, instead of storing extra information for each edit box, I've checked each control of the form to determine which label is connected to the current edit box (indicated by the `Sender` parameter):

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    // if the control is a label
    if (Controls [I] is TLabel) and
      // and the label is connected to the current edit box
      (TLabel(Controls[I]).FocusControl = Sender) then
    // copy the text, leaving off the initial & character
    StatusBar1.SimpleText := 'Enter ' +
      Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;
```

The form's second event handler relates to the first edit box's `OnExit` event. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```
procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
  if EditFirstName.Text = '' then
  begin
    // don't let the user get out
    EditFirstName.SetFocus;
    MessageDlg ('First name is required', mtError, [mbOK], 0);
  end
  else if EditFirstName.Text = 'Admin' then
  begin
    // fill the second edit and jump to the third
    EditLastName.Text := 'Admin';
    EditPassword.SetFocus;
  end;
end;
```

TIP *The CLX version of this example has the same code and is available as the QInFocus program.*

Control Anchors

To let you create a nice, flexible user interface, with controls adapting themselves to the current size of the form, Delphi allows you to determine the relative position of a control with the `Anchors` property. Before this feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom, unless it was aligned to the bottom or right side. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to anchor a button to the bottom-right corner of the form, you place the button in the required position and set its `Anchors` property to [akRight, akBottom]. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

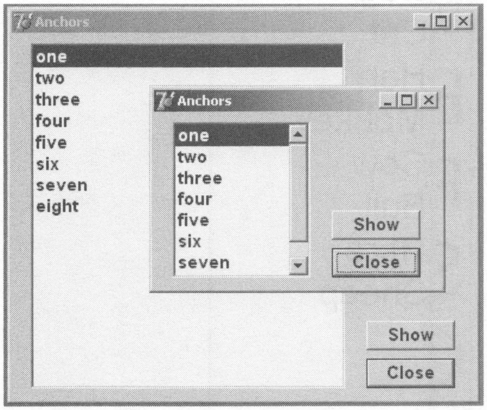
On the other hand, if you place a large component such as a Memo or a ListBox in the middle of a form, you can set its `Anchors` property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

TIP *Anchors, like constraints, work both at design time and at run time. You should set them up as early as possible, to benefit from this feature while you're designing the form as well as at run time.*

As an example of both approaches, you can try the Anchors application, which has two buttons in the bottom-right corner and a list box in the middle. As shown in Figure 5.8, the controls automatically move and stretch as the form size changes. To make this form work properly, you must also set its `Constraints` property; otherwise, if the form becomes too small, the controls can overlap or disappear.

FIGURE 5.8

The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only proper use of the `Anchors` property.



Notice that if you remove all the anchors or two opposite ones (for example, left and right), the resize operations will cause the control to *float* in the form. The control keeps its current size, and the system adds or removes the same number of pixels on each side of it. This anchor can be defined as centered, because if the component is initially in the middle of the form it will keep that position. If you want a centered control you should generally use both opposite anchors, so that if the user makes the form larger, the control size will grow as well. In the case just presented, making the form larger leaves a small control in its center.

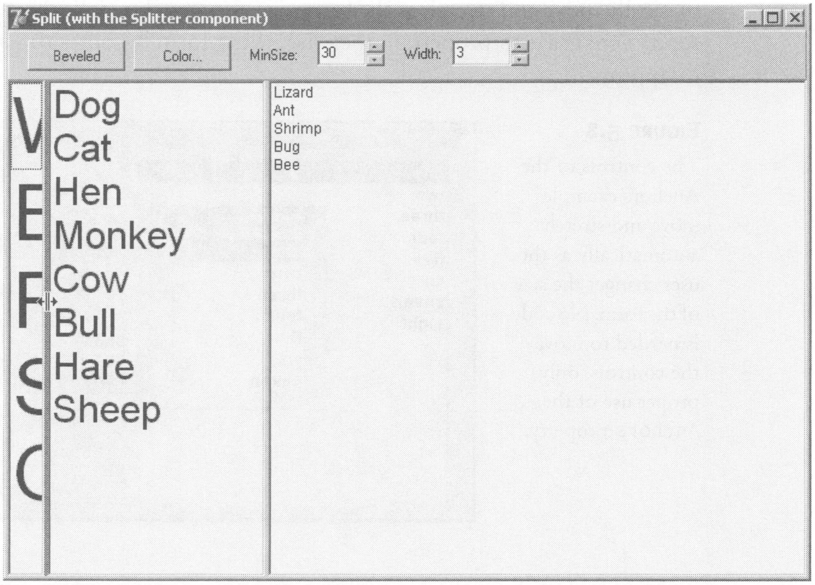
Using the Splitter Component

There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the `Constraints` property of the controls it relates to. As you'll see in the Split1 example, this technique allows you to define maximum and minimum positions for the splitter and the form. To build this example, simply place a ListBox component in a form; then add a Splitter component, a second ListBox, another Splitter, and finally a third ListBox component. The form also has a simple toolbar based on a panel.

By simply placing these two splitter components, you give your form the complete functionality of moving and sizing the controls it hosts at run time. The `Width`, `Beveled`, and `Color` properties of the splitter components determine their appearance, and in the Split1 example you can use the toolbar controls to change them. Another relevant property is `MinSize`, which determines the minimum size of the form's components. During the splitting operation (see Figure 5.9), a line marks the final position of the splitter, but you cannot drag this line beyond a certain limit. The behavior of the Split1 program is not to let controls become too small. An alternative technique is to set the new `AutoSnap` property of the splitter to True. This property will make the splitter hide the control when its size goes below the `MinSize` limit.

FIGURE 5.9

The Split1 example's splitter component determines the minimum size for each control on the form, even those not adjacent to the splitter.



MinSize: 30

Width: 3

I suggest you try using the Split1 program, so that you'll fully understand how the splitter affects its adjacent controls and the other controls of the form. Even if you set the `MinSize` property, a user can reduce the size of the program's entire form to a minimum, hiding some of the list boxes. If you test the Split2 version of the example, you'll get better behavior. In Split2, I've set some `Constraints` for the ListBox controls:

```
object ListBox1: TListBox
  Constraints.MaxHeight = 400
  Constraints.MinHeight = 200
  Constraints.MinWidth = 150
```

The size constraints are applied only as you resize the controls; so, to make this program work satisfactorily, you have to set the `ResizeStyle` property of the two splitters to rsUpdate. This value indicates that the controls' position is updated for every movement of the splitter, not only at the end of the operation. If you select the rsLine or the new rsPattern value, instead, the splitter simply draws a line in the required position, checking the `MinSize` property but not the constraints of the controls.

TIP *When you set the Splitter component's* `AutoSnap` *property to True, the splitter will completely hide the neighboring control when the size of that control is below the minimum set for it in the Splitter component.*

HORIZONTAL SPLITTING

You can also use the Splitter component for horizontal splitting, instead of the default vertical splitting. Basically, you place a component on a form, align it to the top, and then place the splitter on the form. By default, the splitter will be left aligned. Choose the alTop value for the `Align` property, and you're done. You can see a form with a horizontal splitter in the SplitH example. This program has two memo components into which you can open a file, and it has a splitter dividing them, defined as follows:

```
object Splitter1: TSplitter
  Cursor = crVSplit
  Align = alTop
  OnMoved = Splitter1Moved
end
```

The program features a status bar, which keeps track of the current height of the two memo components. It handles the `OnMoved` event of the splitter (the only event of this component) to update the text of the status bar. The same code is executed whenever the form is resized:

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format ('Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

Accelerator Keys

Since Delphi 5, you don't need to enter the `&` character in the `Caption` of a menu item, which provides an automatic accelerator key if you omit one. Delphi's automatic accelerator-key system can also figure out if you have entered conflicting accelerator keys and fix them on the fly. This doesn't mean you should stop adding custom accelerator keys with the `&` character, because the automatic system simply uses the first available letter and doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This feature is controlled by the `AutoHotkeys` property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to maAutomatic; in the pull-downs and menu items, it defaults to maParent, so the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of maAutomatic or maManual.

The engine behind this system is the `RethinkHotkeys` method of the `TMenuItem` class, and the companion `InternalRethinkHotkeys` method. There is also a `RethinkLines` method, which checks whether a pull-down has two consecutive separators or begins or ends with a separator. In all these cases, the separator is automatically removed.

One of the reasons Delphi includes this feature is the support for translations. When you need to translate an application's menu, it is convenient if you don't have to deal with the accelerator keys, or at least if you don't have to worry about whether two items on the same menu conflict. Having a system that can automatically resolve similar problems is definitely an advantage. Another motivation was Delphi's IDE. With all the dynamically loaded packages that install menu items in the IDE main menu or in pop-up menus, and with different packages loaded in different versions of the product, it's next to impossible to get nonconflicting accelerator-key selections in each menu. That is why this mechanism isn't a wizard that does static analysis of your menus at design time; it was created to deal with the real problem of managing menus created dynamically at run time.

WARNING *This feature is certainly handy, but because it is active by default, it can break existing code. I had to modify two of this chapter's program examples, between the Delphi 4 and Delphi 5 edition of the book, just to avoid run-time errors caused by this change. The problem is that I use the caption in the code, and the extra & broke my code. The change was quite simple, though: All I had to do was to set the* `AutoHotkeys` *property of the main menu component to maManual.*

Using the Fly-by Hints

Another common element in toolbars is the *tooltip*, also called *fly-by hint*—text that briefly describes the button currently under the cursor. This text is usually displayed in a yellow box after the mouse cursor has remained steady over a button for a set amount of time. To add hints to a group of buttons or components, simply set the `ShowHints` property of the parent control to True and enter some text for the `Hint` property of each element. You might want to enable the hints for all the components on a form, or all the buttons of a toolbar or panel.

If you want to have more control over how hints are displayed, you can use some of the properties and events of the `Application` object. This global object has, among others, the following properties:

Property	Defines
`HintColor`	The background color of the hint window
`HintPause`	How long the cursor must remain on a component before hints are displayed
`HintHidePause`	How long the hint will be displayed
`HintShortPause`	How long the system should wait to display a hint if another hint has just been displayed

For example, a program might allow a user to customize the hint background color by selecting a specific color with the following code:

```
ColorDialog.Color := Application.HintColor;
if ColorDialog.Execute then
  Application.HintColor := ColorDialog.Color;
```

As an alternative, you can change the hint color by handling the `OnShowHint` property of the `Application` object. This handler can change the hint's color for specific controls. The `OnShowHint` event is used in the CustHint example described in the next section.

CUSTOMIZING THE HINTS

Just as you can add hints to an application's toolbar, you can add hints to forms or to the components of a form. For a large control, the hint will show up near the mouse cursor. In some cases, it is important to know that a program can freely customize how hints are displayed. One thing you can do is to change the value of the properties of the `Application` object, as I mentioned at the end of the last section. To obtain more control over hints, you can customize them even further by assigning a method to the application's `OnShowHint` event. You need to either hook up this event manually or—better—add an ApplicationEvents component to the form and handle its `OnShowHint` event.

The event handler method has some interesting parameters, such as a string with the hint's text, a Boolean flag for its activation, and a `THintInfo` structure with further information, including the control, the hint position, and its color. The parameters are passed by reference, so you have a chance to change them and also modify the values of the `THintInfo` structure; for example, you can change the position of the hint window before it is displayed.

This is what I've done in the CustHint example, which shows the hint for the label at the center of its area.

```
procedure TForm1.ShowHint (var HintStr: string; var CanShow: Boolean;
  var HintInfo: THintInfo);
begin
  with HintInfo do
    // if the control is the label show the hint in the middle
    if HintControl = Label1 then
      HintPos := HintControl.ClientToScreen (Point (
        HintControl.Width div 2, HintControl.Height div 2));
end;
```

The code retrieves the center of the generic control (the `HintInfo.HintControl`) and then converts its coordinates to screen coordinates, applying the `ClientToScreen` method to the control.

You can further update the CustHint example in a different way. The form's ListBox control has some rather long text items, so you might want to display the entire text in a hint while the mouse moves over the item. Setting a single hint for the list box won't do, of course.

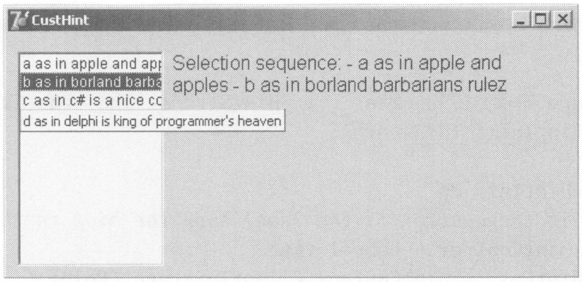
A good solution is to customize the hint system by providing a hint dynamically corresponding to the text of the list box item under the cursor. You also need to indicate to the system which area the hint belongs to, so that by moving over the next line a new hint will be displayed. You accomplish this by setting the `CursorRect` field of the `THintInfo` record, which indicates the area of the component that the cursor can move over without disabling the hint. When the cursor moves outside this area, Delphi hides the hint window. Here is the related code snippet I've added to the `ShowHint` method:

```
else if HintControl = ListBox1 then
  begin
    nItem := ListBox1.ItemAtPos(
      Point (CursorPos.x, CursorPos.Y), True);
    if nItem >= 0 then
    begin
      // set the hint string
      HintStr := ListBox1.Items[nItem];
      // determine area for hint validity
      CursorRect := ListBox1.ItemRect(nItem);
      // display over the item
      HintPos := HintControl.ClientToScreen (Point(
        0, ListBox1.ItemHeight * (nItem - ListBox1.TopIndex)));
    end
    else
      CanShow := False;
  end;
```

The resulting effect is that each line of the list box appears to have a specific hint, as shown in Figure 5.10. The hint position is computed so that it covers the current item text, extending beyond the list box border.

FIGURE 5.10

The ListBox control of the CustHint example shows a different hint, depending on which list item the mouse is over.



a as in apple and app
b as in borland barba
c as in c# is a nice cc
d as in delphi is king of programmer's heaven

Selection sequence: - a as in apple and apples - b as in borland barbarians rulez

Owner-Draw Controls and Styles

In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically, these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In the VCL, the situation is slightly more complex. The components can take care of painting themselves in this case (as in the `TBitBtn` class for bitmap buttons) and possibly activate corresponding events. The system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers. In CLX, some of the controls, such as ListBoxes and ComboBoxes, surface events very similar to Windows owner-draw, but menus lack them. The native approach of Qt is to use styles to determine the graphical behavior of all the controls in the system, of a specific application, or of a given control. I'll introduce styles shortly, later in this section.

NOTE *Most of the Win32 common controls have support for the owner-draw technique, generally called* custom drawing. *You can fully customize the appearance of a ListView, TreeView, TabControl, PageControl, HeaderControl, StatusBar, or ToolBar. The ToolBar, ListView, and TreeView controls also support* advanced *custom drawing, a more fine-tuned drawing capability introduced by Microsoft in the latest versions of the Win32 common controls library. The downside to owner-draw is that when the Windows user interface style changes in the future (and it always does), your owner-draw controls that fit in perfectly with the current user interface styles will look outdated and out of place. Because you are creating a custom user interface, you'll need to keep it updated yourself. By contrast, if you use the standard output of the controls, your applications will automatically adapt to a new version of such controls.*

OWNER-DRAW MENU ITEMS

VCL makes the development of graphical menu items quite simple compared to the traditional approach of the Windows API: You set the `OwnerDraw` property of a menu item component to True and handle its `OnMeasureItem` and `OnDrawItem` events. In the `OnMeasureItem` event, you can determine the size of the menu items. This event handler is activated once for each menu item when the pull-down menu is displayed and has two reference parameters you can set: `Width` and `Height`. In the `OnDrawItem` event, you paint the actual image. This event handler is activated every time the item has to be repainted. This happens when Windows first displays the items and each time the status changes; for example, when the mouse moves over an item, the item should become highlighted.

To paint the menu items, you must consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on. Luckily, the Delphi event passes to the handler the `Canvas` where it should paint, the output rectangle, and the status of the item (selected or not). In the ODMenu example, I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've set the `OwnerDraw` property of the menu and written handlers for some of the menu items. To write a single handler for each event of the three color-related menu items, I've set their `Tag` property to the value of the color in the `OnCreate` event handler of the form. This makes the handler of the items' `OnClick` event quite straightforward:

```
procedure TForm1.ColorClick(Sender: TObject);
begin
  ShapeDemo.Brush.Color := (Sender as TComponent).Tag
end;
```

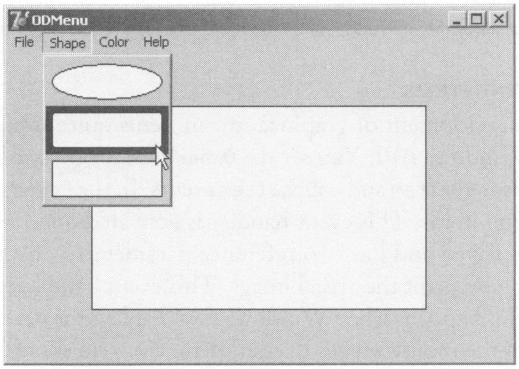
The handler of the `OnMeasureItem` event doesn't depend on the actual items, but uses fixed values (different from the handler of the other pull-down). The most important portion of the code is in the handlers of the `OnDrawItem` events. For the color, you use the value of the tag to paint a rectangle of the given color, as you can see in Figure 5.11. Before doing this, however, you have to fill the background of the menu items (the rectangular area passed as a parameter) with the standard color for the menu (clMenu) or the selected menu items (clHighlight):

```
procedure TForm1.ColorDrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
```

```
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // show the color
  ACanvas.Brush.Color := (Sender as TComponent).Tag;
  InflateRect (ARect, -5, -5);
  ACanvas.Rectangle (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

FIGURE 5.11

The owner-draw menu of the ODMenu example



The three handlers for this event of the Shape pull-down menu items are all different, although they use similar code:

```
procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // draw the ellipse
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

NOTE *To accommodate the increasing number of states in the Windows 2000 user interface style, Delphi includes the* `OnAdvancedDrawItem` *event for menus.*

A LISTBOX OF COLORS

As you have just seen for menus, list boxes have an owner-draw capability, which means a program can paint the items of a list box. The same support is provided for combo boxes and is also available on CLX. To create an owner-draw list box, you set its `Style` property to lbOwnerDrawFixed or lbOwnerDrawVariable. The first value indicates that you will set the height of the list box items by specifying the `ItemHeight` property and that this will be the height of every item. The second owner-draw style indicates a list box with items of different heights; in this case, the component will trigger the `OnMeasureItem` event for each item, to ask the program for their heights.

In the ODList example (and its QODList version), I'll stick with the first, simpler, approach. The example stores color information along with the list box items and then draws the items using those colors (instead of using a single color for the whole list).

The DFM or XFM file of every form, including this one, has a `TextHeight` attribute, which indicates the number of pixels required to display text. You should use this value for the list box's `ItemHeight` property. An alternative solution is to compute this value at run time, so that if you later change the font at design time, you don't have to remember to set the height of the items accordingly.

NOTE *I've just described* `TextHeight` *as an attribute of the form, not a property. It isn't a property, but a local value of the form. If it is not a property, you might ask, how does Delphi save it in the* `DFM` *file? The answer is that Delphi's streaming mechanism is based on properties plus special property clones created by the* `DefineProperties` *method.*

Because `TextHeight` is *not* a property, although it is listed in the form description, you cannot access it directly. Studying the VCL source code, I found that this value is computed by calling a private method of the form: `GetTextHeight`. Because it is private, you cannot call this function. Instead, you can duplicate its code (which is quite simple) in the `FormCreate` method of the form, after selecting the font of the list box:

```
Canvas.Font := ListBox1.Font;
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

Next you add some items to the list box. Because this is a list box of colors, you want to add color names to the `Items` of the list box and the corresponding color values to the `Objects` data storage related to each list item. Instead of adding the two values separately, I've written a procedure to add new items to the list:

```
procedure TODListForm.AddColors (Colors: array of TColor);
var
  I: Integer;
begin
  for I := Low (Colors) to High (Colors) do
    ListBox1.Items.AddObject (ColorToString (Colors[I]), TObject(Colors[I]));
end;
```

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. For each item passed as a parameter, you add the name of the color to the list, and you add its value to the related data by calling the `AddObject` method. To obtain the string corresponding to the color, you call the Delphi `ColorToString` function. It returns a string containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the `TObject` data type (a four-byte reference), as required by the `AddObject` method.

TIP Besides `ColorToString`*, which converts a color value into the corresponding string with the identifier or the hexadecimal value, the Delphi* `StringToColor` *function converts a properly formatted string into a color.*

In the ODList example, this method is called in the form's `OnCreate` event handler (after the height of the items has been set):

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime, clPurple,
  clGray, RGB (213, 23, 123), RGB (0, 0, 0), clAqua, clNavy, clOlive, clTeal]);
```

To compile the CLX version of this code, I've added to it the `RGB` function described earlier in the section "Colors." The code used to draw the items is not particularly complex. You simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  with Control as TListbox do
  begin
    // erase
    Canvas.FillRect(Rect);
    // draw item
    Canvas.Font.Color := TColor (Items.Objects [Index]);
    Canvas.TextOut(Rect.Left, Rect.Top, Listbox1.Items[Index]);
   end;
end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on your part. Moreover, the program allows you to add new items by double-clicking on the list box:

```
procedure TODListForm.ListBox1DblClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    AddColors ([ColorDialog1.Color]);
end;
```

If you try using this capability, you'll notice that some colors you add are turned into color names (one of the Delphi color constants), whereas others are converted into hexadecimal numbers.

ListView and TreeView Controls

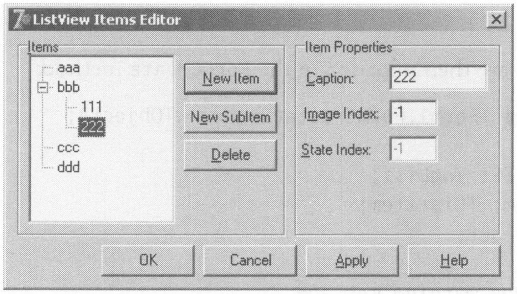
In the earlier section "Opening the Component Toolbox," I introduced the various visual controls you can use to display lists of values. The standard list box and combo box components are very common, but they are often replaced by the more powerful ListView and TreeView controls. These two controls are part of the Win32 common controls, stored in the `ComCtl32.DLL` library. Similar controls are available in Qt and VisualCLX, both on Windows and Linux.

A Graphical Reference List

When you use a ListView component, you can provide bitmaps both indicating the status of the element (for example, the selected item) and describing the contents of the item in a graphical way.

To connect the images to a list or tree, you need to refer to the ImageList component you've already used for the menu images. A ListView can have three image lists: one for the large icons (the `LargeImages` property), one for the small icons (the `SmallImages` property), and one for the state of the items (the `StateImages` property). In the RefList example, I've set the first two properties using two different ImageList components.

Each item of the ListView has an `ImageIndex`, which refers to its image in the list. For this technique to work properly, the elements in the two image lists should follow the same order. When you have a fixed image list, you can add items to it using Delphi's ListView Item Editor, which is connected to the `Items` property. In this editor, you can define items and subitems. The subitems are displayed only in the detailed view (when you set the vsReport value of the `ViewStyle` property) and are connected with the titles set in the `Columns` property:



- aaa
- bbb
  - 111
  - 222
- ccc
- ddd

Caption: 222
Image Index: -1
State Index: -1

WARNING *The ListView control in CLX doesn't have the small and large icon views. In Qt, this type of display is available from another component, the IconView.*

In my RefList example (a simple list of references to books, magazines, CD-ROMs, and websites), the items are stored to a file, because users of the program can edit the contents of the list, which are automatically saved as the program exits. This way, edits made by the user become persistent. Saving and loading the contents of a ListView is not trivial, because the `TListItems` type doesn't have an automatic mechanism to save the data. As an alternative approach, I've copied the data to and from a string list, using a custom format. The string list can then be saved to a file and reloaded with a single command.

The file format is simple, as you can see in the following saving code. For each list item, the program saves the caption on one line, the image index on another line (prefixed by the `@` character), and the subitems on the following lines, indented with a tab character:

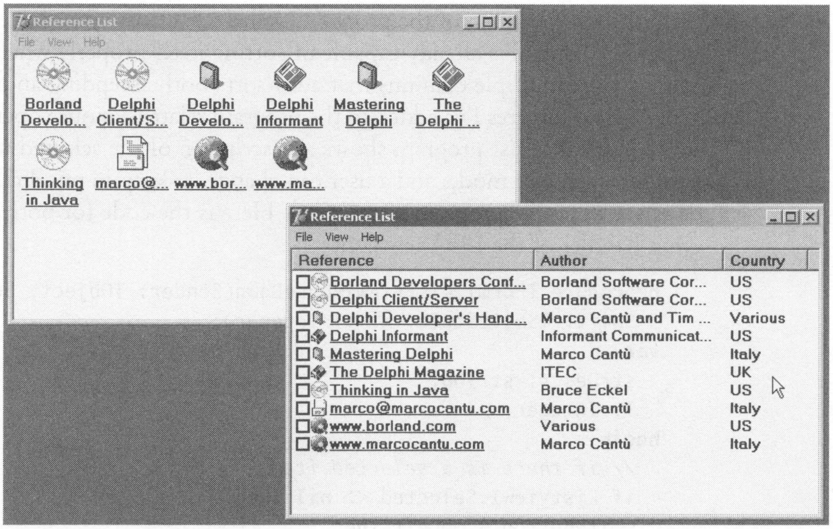
```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I, J: Integer;
  List: TStringList;
begin
  // store the items
  List := TStringList.Create;
  try
    for I := 0 to ListView1.Items.Count - 1 do
```

```
      begin
        // save the caption
        List.Add (ListView1.Items[I].Caption);
        // save the index
        List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
        // save the subitems (indented)
        for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
          List.Add (#9 + ListView1.Items[I].SubItems [J]);
      end;
      List.SaveToFile (ExtractFilePath (Application.ExeName) + 'Items.txt');
    finally
      List.Free;
    end;
  end;
```

The items are then reloaded in the `FormCreate` method:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStringList;
  NewItem: TListItem;
  I: Integer;
begin
  // stops warning message
  NewItem := nil;
  // load the items
  ListView1.Items.Clear;
  List := TStringList.Create;
  try
    List.LoadFromFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
    for I := 0 to List.Count - 1 do
      if List [I][1] = #9 then
        NewItem.SubItems.Add (Trim (List [I]))
      else if List [I][1] = '@' then
        NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
      else
      begin
        // a new item
        NewItem := ListView1.Items.Add;
        NewItem.Caption := List [I];
      end;
  finally
    List.Free;
  end;
end;
```

The program has a menu you can use to choose one of the different views supported by the ListView control and to add check boxes to the items, as in a CheckListBox control. You can see some combinations of these styles in Figure 5.12.



Reference	Author	Country
☐ Borland Developers Conf...	Borland Software Cor...	US
☐ Delphi Client/Server	Borland Software Cor...	US
☐ Delphi Developer's Hand...	Marco Cantù and Tim ...	Various
☐ Delphi Informant	Informant Communicat...	US
☐ Mastering Delphi	Marco Cantù	Italy
☐ The Delphi Magazine	ITEC	UK
☐ Thinking in Java	Bruce Eckel	US
☐ marco@marcocantu.com	Marco Cantù	Italy
☐ www.borland.com	Various	US
☐ www.marcocantu.com	Marco Cantù	Italy

FIGURE 5.12 Different examples of the output of a ListView component in the RefList program, obtained by changing the `ViewStyle` property and adding the check boxes

Another important feature, which is common in the detailed or report view of the control, lets a user sort the items on one of the columns. In the VCL, this technique requires three operations. First, you set the `SortType` property of the ListView to stBoth or stData. This way, the ListView will sort based not on the captions, but by calling the `OnCompare` event for each two items it has to sort.

Second, because you want to sort on each of the columns of the detailed view, you also handle the `OnColumnClick` event (which takes place when the user clicks on the column titles in the detailed view, but only if the `ShowColumnHeaders` property is set to True). Each time a column is clicked, the program saves the number of that column in the form class's `nSortCol` private field:

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  nSortCol := Column.Index;
  ListView1.AlphaSort;
end;
```

Then, in the third step, the sorting code uses either the caption or one of the subitems according to the current sort column:

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem; Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
      Item2.SubItems [nSortCol - 1]);
end;
```

In the CLX version of the program (called QRefList) you don't have to do any of the previous steps. The control is already capable of sorting itself properly when its caption is clicked. You automatically get multiple columns that auto-sort (both ascending and descending).

The final features I've added to the program relate to mouse operations. When the user left-clicks an item, the RefList program shows a description of the selected item. Right-clicking the selected item sets it in edit mode, and a user can change it (keep in mind that the changes will automatically be saved when the program terminates). Here is the code for both operations, in the `OnMouseDown` event handler of the ListView control:

```
procedure TForm1.ListView1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
    begin
      // create and show a description
      strDescr := ListView1.Columns [0].Caption + #9 +
        ListView1.Selected.Caption + #13;
      for I := 1 to ListView1.Selected.SubItems.Count do
        strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
          ListView1.Selected.SubItems [I-1] + #13;
      ShowMessage (strDescr);
    end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
end;
```

Although it is not feature-complete, this example shows some of the potential of the ListView control. I've also activated the "hot-tracking" feature, which lets the list view highlight and underline the item under the mouse. The relevant properties of the ListView can be seen in its textual description:

```
object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
    item
```

```
   Caption = 'Country'
   Width = 80
  end>
 Font.Height = -13
 Font.Name = 'MS Sans Serif'
 Font.Style = [fsBold]
 FullDrag = True
 HideSelection = False
 HotTrack = True
 HotTrackStyles = [htHandPoint, htUnderlineHot]
 SortType = stBoth
 ViewStyle = vsList
 OnColumnClick = ListView1ColumnClick
 OnCompare = ListView1Compare
 OnMouseDown = ListView1MouseDown
end
```

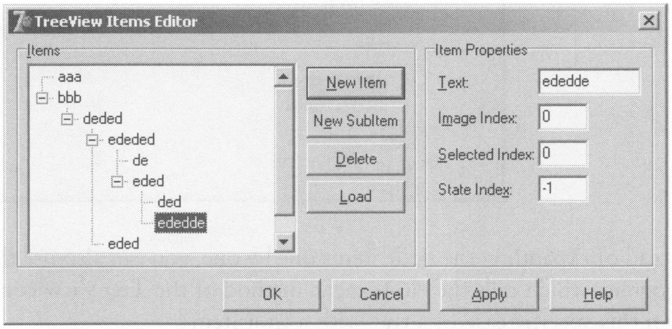
This program is quite interesting, and I'll further extend it in Chapter 9 by adding a dialog box.

To build its CLX version, QRefList, I had to use only one of the image lists and disable the small images and large images menus, because a ListView is limited to the list and report view styles. Large and small icons are available in a different control, called IconView. As previously mentioned, the sorting support was already there, which could have saved most of the code of the example.

A Tree of Data

Now that you've seen an example based on the ListView, let's examine the TreeView control. The TreeView has a user interface that is flexible and powerful (with support for editing and dragging elements). It is also standard, because it is the Windows Explorer user interface. There are properties and various ways to customize the bitmap of each line or each type of line.

To define the structure of the TreeView nodes at design time, you can use the TreeView Items Editor:



Text: ededde
Image Index: 0
Selected Index: 0
State Index: -1

In this case, however, I've decided to load the TreeView data at startup, in a way similar to the last example.

The `Items` property of the TreeView component has many member functions you can use to alter the hierarchy of strings. For example, you can build a two-level tree with the following lines:

```
var
  Node: TTreeNode;
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

Using the `Add` and `AddChild` methods, you can build a complex structure at run time. To load the information, you can again use a StringList at run time, load a text file with the information, and parse it.

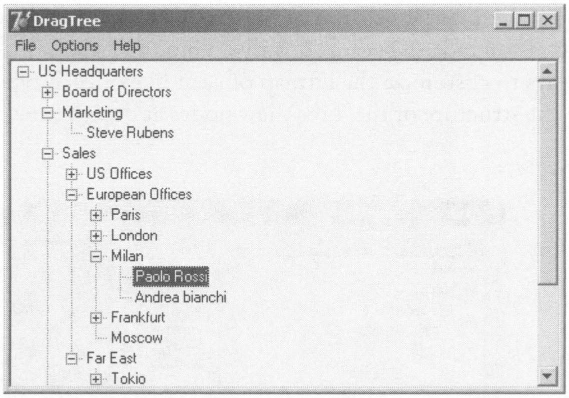
However, because the TreeView control has a `LoadFromFile` method, the DragTree and QDragTree examples use the following simpler code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.LoadFromFile (ExtractFilePath (Application.ExeName) +
    'TreeText.txt');
end;
```

The `LoadFromFile` method loads the data in a string list and checks the level of each item by looking at the number of tab characters. (If you are curious, see the `TTreeStrings.GetBufStart` method, which you can find in the ComCtrls unit in the VCL source code included in Delphi.) The data I've prepared for the TreeView is the organizational chart of a multinational company, as you can see in Figure 5.13.

FIGURE 5.13

The DragTree example after loading the data and expanding the branches


DragTree
File
Options
Help
US Headquarters
Board of Directors
Marketing
Steve Rubens
Sales
US Offices
European Offices
Paris
London
Milan
Paolo Rossi
Andrea bianchi
Frankfurt
Moscow
Far East
Tokio

Instead of expanding the node items one by one, you can also use the File ➢ Expand All menu of this program, which calls the `FullExpand` method of the TreeView control or executes the equivalent code (in this specific case of a tree with a root item):

```
TreeView1.Items [0].Expand(True);
```

Besides loading the data, the program saves the data when it terminates, making the changes persistent. It also has a few menu items to customize the font of the TreeView control and change some other simple settings. The specific feature I've implemented in this example is support for dragging items and entire subtrees. I've set the `DragMode` property of the component to dmAutomatic and written the event handlers for the `OnDragOver` and `OnDragDrop` events.

In the first of the two handlers, the program makes sure the user is not trying to drag an item over a child item (which would be moved along with the item, leading to an infinite recursion):

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
  begin
    Accept := True;
    // determines source and target
    SourceNode := TreeView1.Selected;
    // look up the target parent chain
    while (TargetNode.Parent <> nil) and (TargetNode <> SourceNode) do
      TargetNode := TargetNode.Parent;
    // if source is found
    if TargetNode = SourceNode then
      // do not allow dragging over a child
      Accept := False;
  end
  else
    Accept := False;
end;
```

The effect of this code is that (except for the particular case you need to disallow) a user can drag a TreeView item over another item. Writing the code for moving the items is simple, because the TreeView control provides support for this operation through the `TTreeNode` class's `MoveTo` method:

```
procedure TForm1.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  if TargetNode <> nil then
  begin
    SourceNode := TreeView1.Selected;
    SourceNode.MoveTo (TargetNode, naAddChildFirst);
    TargetNode.Expand (False);
    TreeView1.Selected := TargetNode;
  end;
end;
```

NOTE *Among the demos shipping with Delphi is an interesting one showing a custom-draw TreeView control. The example is in the* `CustomDraw` *subdirectory.*

THE PORTABLE VERSION OF DRAGTREE

Because I use this program in numerous porting demonstrations, I've built a version you can compile as a native VCL application with Delphi and as a CLX application with Kylix. This is different from most other programs in this book, including the previous version of this same example, which can be ported to Delphi by using VisualCLX and also Qt on Windows. Following a different path once in a while can be instructive.

The first thing I had to do was use two different sets of `uses` statements, using conditional compilation. The unit of the PortableDragTree example begins as follows:

```
unit TreeForm;

interface

uses
  SysUtils, Classes,

{$IFDEF LINUX}
  Qt, Libc, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls, QComCtrls, QMenus, QTypes, QGrids;
{$ENDIF}

{$IFDEF MSWINDOWS}
  Windows, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls, Menus, Grids;
{$ENDIF}
```

A similar conditional directive is used in the initial portion of the implementation section, to include the proper resource file for the form (the two resource files are different):

```
{$IFDEF LINUX}
  {$R *.xfm}
{$ENDIF}

{$IFDEF MSWINDOWS}
  {$R *.dfm}
{$ENDIF}
```

I've omitted some of the Windows-specific features anyway, so the only difference in code is in the `FormCreate` method. The program loads the data file from the user's default folder, not the same folder as the executable. Depending on the operating system, the user's folder is the home directory (and the hidden file has a starting dot) or the specific My Documents area (available with a special API call):

```
procedure TForm1.FormCreate(Sender: TObject);
var
  path: string;
```

```
begin
{$IFDEF LINUX}
  filename := GetEnvironmentVariable('HOME') +
    '/.TreeText.txt';
{$ELSE}
  SetLength (path, 100);
  ShGetSpecialFolderPath (Handle, PChar(path),
    CSIDL_PERSONAL, False);
  path := PChar (path); // fix string length
  filename := path + '\TreeText.txt';
{$ENDIF}
  TreeView1.LoadFromFile (filename);
end;
```

Custom Tree Nodes

Delphi 6 added a few features to the TreeView controls, including multiple selection (see the `MultiSelect` and `MultiSelectStyle` properties and the `Selections` array), improved sorting, and several new events. The key improvement, however, is letting the programmer determine the class of the tree view's node items. Having custom node items implies the ability to attach custom data to the nodes in a simple, object-oriented way. To support this technique, there is a new `AddNode` method for the `TTreeItems` class and a new specific event, `OnCreateNodesClass`. In the handler for this event, you return the class of the object to be created, which must inherit from `TTreeNode`.

This is a very common technique, so I've built an example to discuss it in detail. The CustomNodes example doesn't focus on a real-world case, but it shows a rather complex situation in which two different custom tree node classes are derived one from the other. The base class adds an `ExtraCode` property, mapped to virtual methods, and the subclass overrides one of these methods. For the base class, the `GetExtraCode` function simply returns the value; for the derived class, the value is multiplied to the parent node value. Here are the classes and this second method:

```
type
  TMyNode = class (TTreeNode)
  private
    FExtraCode: Integer;
  protected
    procedure SetExtraCode(const Value: Integer); virtual;
    function GetExtraCode: Integer; virtual;
  public
    property ExtraCode: Integer read GetExtraCode write SetExtraCode;
  end;

  TMySubNode = class (TMyNode)
  protected
    function GetExtraCode: Integer; override;
  end;
```

```
function TMySubNode.GetExtraCode: Integer;
begin
  Result := fExtraCode * (Parent as TMyNode).ExtraCode;
end;
```

With these custom tree node classes available, the program creates a tree of items, using the first type for the first-level nodes and the second class for the other nodes. Because you have only one `OnCreateNodeClass` event handler, the program uses the class reference stored in a private field of the form (`CurrentNodeClass` of type `TTreeNodeClass`):

```
procedure TForm1.TreeView1CreateNodeClass(Sender: TCustomTreeView;
  var NodeClass: TTreeNodeClass);
begin
  NodeClass := CurrentNodeClass;
end;
```

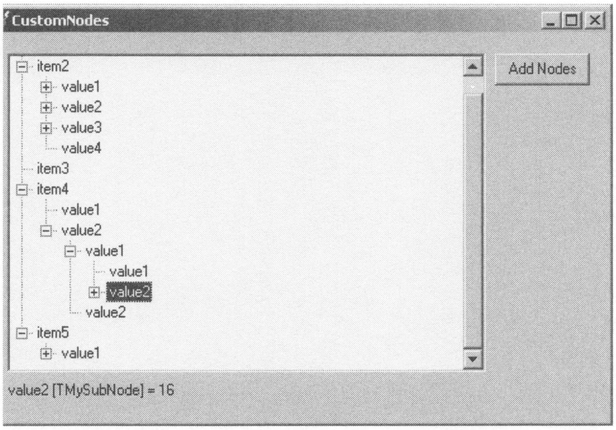
The program sets this class reference before creating nodes of each type—for example, with code like the following:

```
var
  MyNode: TMyNode;
begin
  CurrentNodeClass := TMyNode;
  MyNode := TreeView1.Items.AddChild (nil, 'item' + IntToStr (nValue))
    as TMyNode;
  MyNode.ExtraCode := nValue;
```

Once the entire tree has been created, when the user selects an item, you can cast its type to `TMyNode` and access the extra properties (but also methods and data):

```
procedure TForm1.TreeView1Click(Sender: TObject);
var
  MyNode: TMyNode;
begin
  MyNode := TreeView1.Selected as TMyNode;
  Label1.Caption := MyNode.Text + ' [' + MyNode.ClassName + '] = ' +
    IntToStr (MyNode.ExtraCode);
end;
```

This is the code used by the CustomNodes example to display the description of the selected node in a label, as you can see in Figure 5.14. Note that when you select an item within the tree, its value is multiplied for that of each parent node. There are certainly easier ways to obtain this effect, but having a tree view with item objects created from different classes of a hierarchy provides an object-oriented structure upon which you can base some very complex code.


CustomNodes
Add Nodes
item2
value1
value2
value3
value4
item3
item4
value1
value2
value1
value1
value2
value2
item5
value1

value2 [TMySubNode] = 16

FIGURE 5.14 The CustomNodes example has a tree view with node objects based on different custom classes, thanks to the `OnCreateNodesClass` event.

What's Next?

In this chapter, we have explored the foundations of the libraries available in Delphi for building user interfaces: the native-Windows VCL and the Qt-based CLX. I've discussed the `TControl` class, its properties, and its most important derived classes.

We explored some of the basic components available in Delphi, looking at both libraries. These components correspond to the standard Windows controls and some of the common controls, and they are extremely common in applications. You've also seen how to create main menus and pop-up menus and how to add extra graphics to some of these controls.

The next step is to explore in depth the elements of a complete user interface, discussing action lists and the Action Manager, and building some simple but complete examples. This is the topic of Chapter 6; then, Chapter 7 is devoted to forms.

Chapter 6

Building the User Interface

In Chapter 5, I discussed the core concepts of the `TControl` class and its derived classes in the VCL and VisualCLX libraries. Then, I provided a rapid tour of the key controls you can use to build a user interface, including editing components, lists, range selectors, and more. This chapter discusses other controls used to define the overall design of a form, such as the PageControl and TabControl. After these components, I'll introduce toolbars and status bars, including some slightly advanced features. This will give you the foundation material for the rest of the chapter, which covers actions and the Action Manager architecture.

Modern Windows applications usually have multiple ways of giving a command, including menu items, toolbar buttons, shortcut menus, and so on. To separate the actual commands a user can give from their multiple representations in the user interface, Delphi uses the concept of *actions*. In recent Delphi versions, this architecture has been extended to make the construction of the user interface on top of actions totally visual. You can now also let program users customize this interface easily, as is true in many professional programs. Finally, Delphi 7 adds to the visual controls supporting the Action Manager architecture a better and more modern UI, supporting the XP look and feel. On Windows XP, you can create applications that adapt themselves to the active theme, thanks to a lot of new internal VCL code.

This chapter covers the following:

- Multipage forms
- Pages and tabs
- ToolBars and a StatusBars
- Themes and styles
- Actions and ActionLists
- Predefined actions in Delphi
- The ControlBar and CoolBar components
- Docking toolbars and other controls
- The Action Manager architecture

Multiple-Page Forms

When you need to display a lot of information and controls in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages. You can use two controls to build a multiple-page application in Delphi:

- The *PageControl* component has tabs on one side and multiple pages (similar to panels) covering the rest of its surface. There is one page per tab, so you can simply place components on each page to obtain the proper effect both at design time and at run time.
- The *TabControl* has only the tab portion but offers no pages to hold the information. In this case, you'll want to use one or more components to mimic the *page change* operation, or you can place different forms within the tabs to simulate the pages.

A third related class, the TabSheet, represents a single page of the PageControl. This is not a stand-alone component and is not available on the Component Palette. You create a TabSheet at design time by using the shortcut menu of the PageControl or at run time by using methods of the same control.

NOTE *Delphi still includes (in the Win 3.1 tab of the Component Palette) the Notebook, TabSet, and TabbedNotebook components introduced in 32-bit versions (that is, since Delphi 2). For any other purpose, the PageControl and TabControl components, which encapsulate Win32 common controls, provide a more modern user interface. In 32-bit versions of Delphi, the TabbedNotebook component was reimplemented using the Win32 PageControl internally, to reduce the code size and update the look.*

PageControls and TabSheets

As usual, instead of duplicating the Help system's list of properties and methods for the PageControl component, I've built an example that stretches the control's capabilities and allows you to change its behavior at run time. The example, called Pages, has a PageControl with three pages. The structure of the PageControl and the other key components appears in Listing 6.1.

LISTING 6.1: KEY PORTIONS OF THE DFM OF THE PAGES EXAMPLE

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    HotTrack = True
    Images = ImageList1
    MultiLine = True
    object TabSheet1: TTabSheet
      Caption = 'Pages'
      object Label3: TLabel
      object ListBox1: TListBox
    end
    object TabSheet2: TTabSheet
      Caption = 'Tabs Size'
      ImageIndex = 1
      object Label1: TLabel
      // other controls
    end
    object TabSheet3: TTabSheet
      Caption = 'Tabs Text'
      ImageIndex = 2
      object Memo1: TMemo
        Anchors = [akLeft, akTop, akRight, akBottom]
        OnChange = Memo1Change
      end
      object BitBtnChange: TBitBtn
        Anchors = [akTop, akRight]
        Caption = '&Change'
      end
    end
  end
  object BitBtnPrevious: TBitBtn
    Anchors = [akRight, akBottom]
    Caption = '&Previous'
    OnClick = BitBtnPreviousClick
  end
  object BitBtnNext: TBitBtn
    Anchors = [akRight, akBottom]
    Caption = '&Next'
    OnClick = BitBtnNextClick
  end
  object ImageList1: TImageList
    Bitmap = {...}
  end
end
```

Notice that the tabs are connected to the bitmaps provided by an ImageList control and that some controls use the `Anchors` property to remain at a fixed distance from the right or bottom borders of the form. Even if the form doesn't support resizing (this would have been far too complex to set up with so many controls), the positions can change when the tabs are displayed on multiple lines (simply increase the length of the captions) or on the left side of the form.

Each TabSheet object has its own `Caption`, which is displayed as the sheet's tab. At design time, you can use the shortcut menu to create new pages and to move between pages. You can see the shortcut menu of the PageControl component in Figure 6.1, together with the first page. This page holds a list box and a small caption, and it shares two buttons with the other pages.

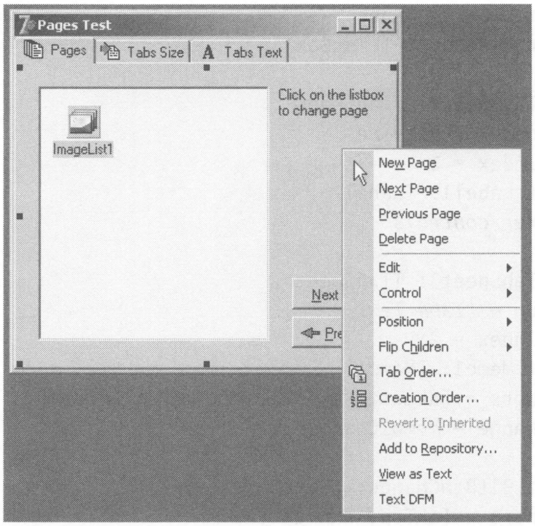

Pages Test
Pages
Tabs Size
Tabs Text
Click on the listbox to change page
ImageList1
Next
Pre
New Page
Next Page
Previous Page
Delete Page
Edit
Control
Position
Flip Children
Tab Order...
Creation Order...
Revert to Inherited
Add to Repository...
View as Text
Text DFM

FIGURE 6.1 The first sheet of the PageControl of the Pages example, with its shortcut menu

If you place a component on a page, it is available only in that page. How can you have the same component (in this case, two bitmap buttons) in each page, without duplicating it? Simply place the component on the form, outside the PageControl (or before aligning it to the client area), and then move it in front of the pages, calling the control ➢ Bring To Front command from the form's shortcut menu. The two buttons I've placed in each page can be used to move back and forth between the pages and are an alternative to using the tabs. Here is the code associated with one of them:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.SelectNextPage (True);
end;
```

The other button calls the same procedure, passing False as its parameter to select the previous page. Notice that there is no need to check whether you are on the first or last page, because the `SelectNextPage` method considers the last page to be the one before the first and will move you directly between those two pages.

Now let's focus on the first page again. It has a list box, which at run time will hold the names of the tabs. If a user clicks an item in this list box, the current page changes. This is the third method available to change pages (after the tabs and the Next and Previous buttons). The list box is filled in the `FormCreate` method, which is associated with the `OnCreate` event of the form and copies the caption of each page (the `Page` property stores a list of TabSheet objects):

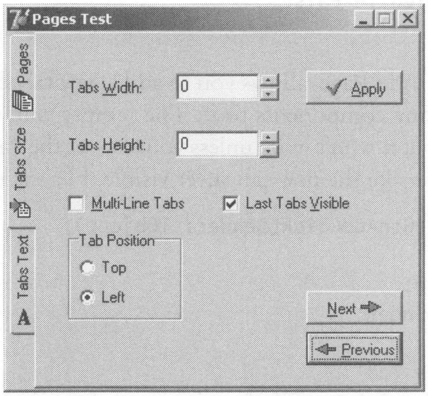
```
for I := 0 to PageControl1.PageCount - 1 do
  ListBox1.Items.Add (PageControl1.Pages.Caption);
```

When you click a list item, you can select the corresponding page:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage := PageControl1.Pages [ListBox1.ItemIndex];
end;
```

The second page hosts two edit boxes (connected with two UpDown components), two check boxes, and two radio buttons, as you can see in Figure 6.2. The user can input a number (or choose it by clicking the up or down buttons with the mouse or pressing the Up or Down arrow key while the corresponding edit box has the focus), check the boxes and the radio buttons, and then click the Apply button to make the changes:

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
  // set tab width, height, and lines
  PageControl1.TabWidth := StrToInt (EditWidth.Text);
  PageControl1.TabHeight := StrToInt (EditHeight.Text);
  PageControl1.MultiLine := CheckBoxMultiLine.Checked;
  // show or hide the last tab
  TabSheet3.TabVisible := CheckBoxVisible.Checked;
  // set the tab position
  if RadioButton1.Checked then
    PageControl1.TabPosition := tpTop
  else
    PageControl1.TabPosition := tpLeft;
end;
```



Tabs Width: 0
Tabs Height: 0
☐ Multi-Line Tabs
☑ Last Tabs Visible
Tab Position
○ Top
● Left

FIGURE 6.2 The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control.

With this code, you can change the width and height of each tab (remember that `0` means the size is computed automatically from the space taken by each string). You can choose to have either multiple lines of tabs or two small arrows to scroll the tab area, and you can move the tabs to the left side of the window. The control also lets you place tabs on the bottom or on the right, but this program doesn't allow that, because it would make the placement of the other controls quite complex.

You can also hide the last tab on the PageControl, which corresponds to the `TabSheet3` component. If you hide one of the tabs by setting its `TabVisible` property to False, you cannot reach that tab by clicking on the Next and Previous buttons, which are based on the `SelectNextPage` method. Instead, you should use the `FindNextPage` function, which will select that page even if the tab won't become visible. A call of `FindNextPage` method is shown in the following new version of the Next button's `OnClick` event handler:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.ActivePage := PageControl1.FindNextPage (
    PageControl1.ActivePage, True, False);
end;
```

The last page has a memo component, again with the names of the pages (added in the `FormCreate` method). You can edit the names of the pages and click the Change button to change the text of the tabs, but only if the number of strings matches the number of tabs:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
  if Memo1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please', mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption := Memo1.Lines [I];
  BitBtnChange.Enabled := False;
end;
```

Finally, the last button, Add Page, allows you to add a new tab sheet to the page control, although the program doesn't add any components to it. The (empty) tab sheet object is created using the page control as its owner, but it won't work unless you also set the `PageControl` property. Before doing this, however, you should make the new tab sheet visible. Here is the code:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);
var
  strCaption: string;
  NewTabSheet: TTabSheet;
begin
  strCaption := 'New Tab';
  if InputQuery ('New Tab', 'Tab Caption', strCaption) then
  begin
    // add a new empty page to the control
    NewTabSheet := TTabSheet.Create (PageControl1);
    NewTabSheet.Visible := True;
    NewTabSheet.Caption := strCaption;
    NewTabSheet.PageControl := PageControl1;
    PageControl1.ActivePage := NewTabSheet;
    // add it to both lists
    Memo1.Lines.Add (strCaption);
    ListBox1.Items.Add (strCaption);
  end;
end;
```

TIP *Whenever you write a form based on a PageControl, remember that the first page displayed at run time is the page you were in before the code was compiled. For example, if you are working on the third page and then compile and run the program, it will start with that page. A common way to solve this problem is to add a line of code in the* `FormCreate` *method to set the PageControl or notebook to the first page. This way, the current page at design time doesn't determine the initial page at run time.*

An Image Viewer with Owner-Draw Tabs

You can also use the TabControl and a dynamic approach, as described in the last example, in more general (and simpler) cases. Every time you need multiple pages that all have the same type of content, instead of replicating the controls in each page, you can use a TabControl and change its contents when a new tab is selected. This is what I've done in the multiple-page bitmap viewer example, called BmpViewer. The image that appears in the TabControl of this form, aligned to the whole client area, depends on the selection in the tab above it (as you can see in Figure 6.3).

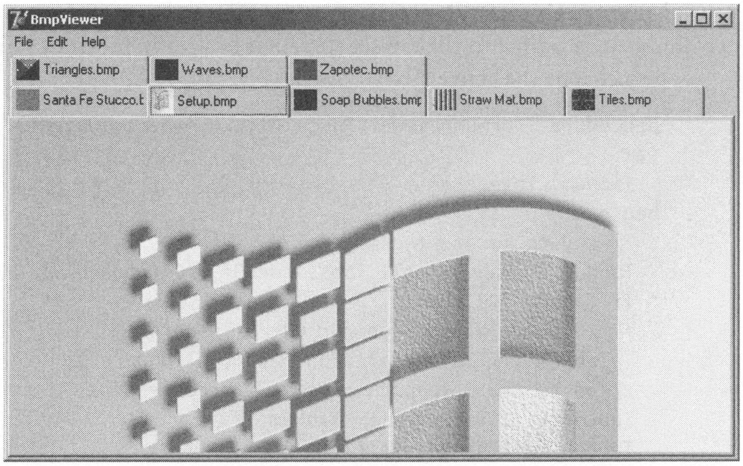


FIGURE 6.3 The interface of the bitmap viewer in the BmpViewer example. Notice the owner-draw tabs.

At the beginning, the TabControl is empty. After selecting File ➢ Open, the user can choose various files in the File Open dialog box, and the array of strings with the names of the files (the `Files` property of the `OpenDialog1` component) is added to the tabs (the `Tabs` property of `TabControl1`):

```
procedure TFormBmpViewer.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    TabControl1.Tabs.AddStrings (OpenDialog1.Files);
    TabControl1.TabIndex := 0;
    TabControl1Change (TabControl1);
  end;
end;
```

WARNING *The* `Tabs` *property of a TabControl in CLX is a collection, whereas in the VCL it is simply a string list.*

After you display the new tabs, you have to update the image so that it matches the first tab. To accomplish this, the program calls the method connected with the `OnChange` event of the TabControl, which loads the file corresponding to the current tab in the image component:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (TabControl1.Tabs [TabControl1.TabIndex]);
end;
```

This example works, unless you select a file that doesn't contain a bitmap. The program will warn the user with a standard exception, ignore the file, and continue its execution.

The program also lets you paste the bitmap on the Clipboard (without immediately getting it, but only adding a tab that will perform the actual paste operation when selected) and copy the current bitmap to it. Clipboard support is available in Delphi via the global `Clipboard` object defined in the ClipBrd unit. For copying or pasting bitmaps, you can use the `Assign` method of the `TClipboard` and `TBitmap` classes. When you select the Edit ➢ Paste command in the example, a new tab named Clipboard is added to the tab set (unless it is already present). Then the number of the new tab is used to change the active tab:

```
procedure TFormBmpViewer.Paste1Click(Sender: TObject);
var
  TabNum: Integer;
begin
  // try to locate the page
  TabNum := TabControl1.Tabs.IndexOf ('Clipboard');
  if TabNum < 0 then
    // create a new page for the Clipboard
    TabNum := TabControl1.Tabs.Add ('Clipboard');
  // go to the Clipboard page and force repaint
  TabControl1.TabIndex := TabNum;
  TabControl1Change (Self);
end;
```

The Edit ➢ Copy operation is as simple as copying the bitmap currently in the image control:

```
Clipboard.Assign (Image1.Picture.Graphic);
```

To account for the possible presence of the Clipboard tab, the code of the `TabControl1Change` method becomes:

```
procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
var
  TabText: string;
begin
  Image1.Visible := True;
  TabText := TabControl1.Tabs [TabControl1.TabIndex];
  if TabText <> 'Clipboard' then
```

```
    // load the file indicated in the tab
    Image1.Picture.LoadFromFile (TabText)
  else
    {if the tab is 'Clipboard' and a bitmap
    is available in the clipboard}
    if Clipboard.HasFormat (cf_Bitmap) then
      Image1.Picture.Assign (Clipboard)
    else
    begin
      // else remove the clipboard tab
      TabControl1.Tabs.Delete (TabControl1.TabIndex);
      if TabControl1.Tabs.Count = 0 then
        Image1.Visible := False;
    end;
```

This program pastes the bitmap from the Clipboard each time you change the tab. The program stores only one image at a time, and it has no way to store the Clipboard bitmap. However, if the Clipboard content changes and the bitmap format is no longer available, the Clipboard tab is automatically deleted (as you can see in the previous listing). If no more tabs are left, the Image component is hidden.

An image can also be removed using either of two menu commands: Cut or Delete. Cut removes the tab after making a copy of the bitmap to the Clipboard. In practice, the `Cut1Click` method does nothing besides call the `Copy1Click` and `Delete1Click` methods. The `Copy1Click` method is responsible for copying the current image to the Clipboard; `Delete1Click` simply removes the current tab. Here is their code:

```
procedure TFormBmpViewer.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign (Image1.Picture.Graphic);
end;

procedure TFormBmpViewer.Delete1Click(Sender: TObject);
begin
  with TabControl1 do
  begin
    if TabIndex >= 0 then
      Tabs.Delete (TabIndex);
    if Tabs.Count = 0 then
      Image1.Visible := False;
  end;
end;
```

One of the special features of the example is that the TabControl has the `OwnerDraw` property set to True. This means the control won't paint the tabs (which will be empty at design time) but will instead have the application do this, by calling the `OnDrawTab` event. In its code, the program displays the text vertically centered, using the `DrawText` API function. The text displayed is not the entire file path but only the filename. Then, if the text is not *None*, the program reads the bitmap the tab refers to and paints a small version of it in the tab itself. To accomplish this, the program uses the `TabBmp` object, which is of type `TBitmap` and is created and destroyed along with the form. The program also uses the `BmpSide` constant to position the bitmap and the text properly:

```
procedure TFormBmpViewer.TabControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
 TabText: string;
 OutRect: TRect;
begin
  TabText := TabControl1.Tabs [TabIndex];
  OutRect := Rect;
  InflateRect (OutRect, -3, -3);
  OutRect.Left := OutRect.Left + BmpSide + 3;
  DrawText (Control.Canvas.Handle, PChar (ExtractFileName (TabText)),
    Length (ExtractFileName (TabText)), OutRect,
    dt_Left or dt_SingleLine or dt_VCenter);
  if TabText = 'Clipboard' then
    if Clipboard.HasFormat (cf_Bitmap) then
      TabBmp.Assign (Clipboard)
    else
      TabBmp.FreeImage
  else
    TabBmp.LoadFromFile (TabText);
  OutRect.Left := OutRect.Left - BmpSide - 3;
  OutRect.Right := OutRect.Left + BmpSide;
  Control.Canvas.StretchDraw (OutRect, TabBmp);
end;
```

The program has also support for printing the current bitmap, after showing a page preview form in which the user can select the proper scaling. This extra portion of the program I built for earlier editions of the book is not discussed in detail, but I've left the code in the program so you can examine it.

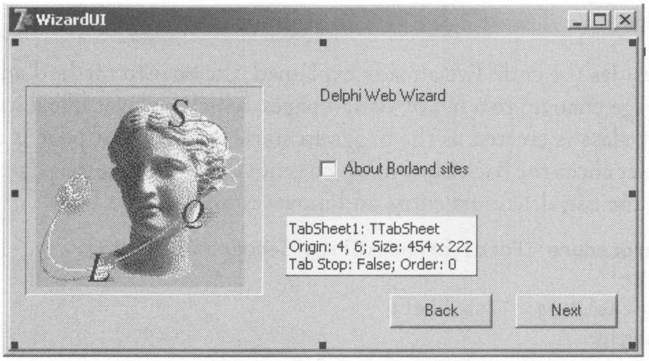
The User Interface of a Wizard

Just as you can use a TabControl without pages, you can also take the opposite approach and use a PageControl without tabs. Now I'll focus on the development of the user interface of a wizard. In a wizard, you direct the user through a sequence of steps, one screen at a time, and at each step you typically offer the choice of proceeding to the next step or going back to correct input entered in a previous step. Instead of tabs that can be selected in any order, wizards typically offer Next and Back buttons to navigate. This won't be a complex example; its purpose is just to give you a few guidelines. The example is called WizardUI.

The starting point is to create a series of pages in a PageControl and set the `TabVisible` property of each TabSheet to False (while keeping the `Visible` property set to True). Since Delphi 5, you can also hide the tabs at design time. In this case, you'll need to use the shortcut menu of the page control, the Object Inspector's combo box, or the Object Tree View to move to another page, instead of the tabs. But why don't you want to see the tabs at design time? You can place controls on the pages and then place extra controls in front of the pages (as I've done in the example), without seeing their relative positions change at run time. You might also want to remove the useless captions from the tabs; they take up space in memory and in the resources of the application.

In the first page, I've placed an image and a bevel control on one side and some text, a check box, and two buttons on the other side. Actually, the Next button is inside the page, and the Back button is over it (and is shared by all the pages). You can see this first page at design time in Figure 6.4. The following pages look similar, with a label, check boxes, and buttons on the right side and nothing on the left.

FIGURE 6.4
The first page of the WizardUI example at design time



TabSheet1: TTabSheet
Origin: 4, 6; Size: 454 x 222
Tab Stop: False; Order: 0

When you click the Next button on the first page, the program looks at the status of the check box and decides which page is the following one. I could have written the code like this:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  BtnBack.Enabled := True;
  if CheckInprise.Checked then
    PageControl1.ActivePage := TabSheet2
  else
    PageControl1.ActivePage := TabSheet3;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

After enabling the common Back button, the program changes the active page and finally moves the graphical portion to the new page. Because this code has to be repeated for each button, I've placed it in a method after adding a couple of extra features. This is the code:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  if CheckInprise.Checked then
    MoveTo (TabSheet2)
  else
    MoveTo (TabSheet3);
end;
```

```
procedure TForm1.MoveTo(TabSheet: TTabSheet);
begin
  // add the last page to the list
  BackPages.Add (PageControl1.ActivePage);
  BtnBack.Enabled := True;
  // change page
  PageControl1.ActivePage := TabSheet;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

Besides the code I've already explained, the `MoveTo` method adds the last page (the one before the page change) to a list of visited pages, which behaves like a stack. The `BackPages` object of the `TList` class is created as the program starts, and the last page is always added to the end. When the user clicks the Back button, which is not dependent on the page, the program extracts the last page from the list, deletes its entry, and moves to that page:

```
procedure TForm1.btnBackClick(Sender: TObject);
var
  LastPage: TTabSheet;
begin
  // get the last page and jump to it
  LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
  PageControl1.ActivePage := LastPage;
  // delete the last page from the list
  BackPages.Delete (BackPages.Count - 1);
  // eventually disable the back button
  BtnBack.Enabled := not (BackPages.Count = 0);
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

With this code, the user can move back several pages until the list is empty, at which point you disable the Back button. You need to deal with a complication: While moving from a particular page, you know which pages are "next" and "previous," but you don't know which page you we came from, because there can be multiple paths to reach a page. Only by keeping track of the movements with a list can you reliably go back.

The rest of the program code, which simply shows some website addresses, is very simple. The good news is that you can reuse the navigational structure of this example in your own programs and modify only the graphical portion and the content of the pages. Because most of the programs' labels show HTTP addresses, a user can click a label to open the default browser showing that page. You accomplish this by extracting the HTTP address from the label and calling the `ShellExecute` function:

```
procedure TForm1.LabelLinkClick(Sender: TObject);
var
  Caption, StrUrl: string;
```

```
begin
  Caption := (Sender as TLabel).Caption;
  StrUrl := Copy (Caption, Pos ('http://', Caption), 1000);
  ShellExecute (Handle, 'open', PChar (StrUrl), '', '', sw_Show);
end;
```

This method is hooked to the `OnClick` event of many labels on the form, which have been turned into *links* by setting the `Cursor` to a hand. This is one of the labels:

```
object Label2: TLabel
  Cursor = crHandPoint
  Caption = 'Main site: http://www.borland.com'
  OnClick = LabelLinkClick
end
```

The ToolBar Control

To create a toolbar, Delphi includes a specific component that encapsulates the corresponding Win32 common control or the corresponding Qt widget in VisualCLX. This component provides a toolbar with its own buttons, and it has many advanced capabilities. You place the component on a form and then use the component editor (the shortcut menu activated by a right-click) to create buttons and separators.

The toolbar is populated with objects of the `TToolButton` class. These objects have a fundamental property, `Style`, which determines their behavior:

- The tbsButton style indicates a standard pushbutton.
- The tbsCheck style indicates a button with the behavior of a check box, or that of a radio button if the button is grouped with the others in its block (determined by the presence of separators).
- The tbsDropDown style indicates a drop-down button (a sort of combo box). The drop-down portion can be easily implemented in Delphi by connecting a PopupMenu control to the `DropdownMenu` property of the control.
- The tbsSeparator and tbsDivider styles indicate separators with no or different vertical lines (depending on the `Flat` property of the toolbar).

To create a graphic toolbar, you can add an ImageList component to the form, load some bitmaps into it, and then connect the ImageList with the `Images` property of the toolbar. By default, the images will be assigned to the buttons in the order they appear, but you can change this behavior quite easily by setting the `ImageIndex` property of each toolbar button. You can prepare further ImageLists for special button conditions and assign them to the `DisabledImages` and `HotImages` properties of the toolbar. The first group is used for the disabled buttons; the second is used for the button currently under the mouse.

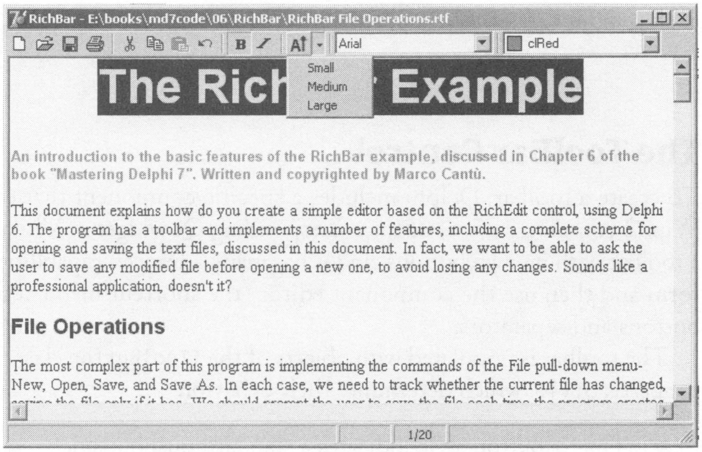
NOTE *In a nontrivial application, you will generally create toolbars using an ActionList or the recent Action Manager architecture, discussed later in this chapter. In this case, you'll attach little behavior to the toolbar buttons, because their properties and events will be managed by the action components. Moreover, you'll end up using a toolbar of the specific* `TActionToolBar` *class.*

The RichBar Example

As an example of the use of a toolbar, I've built the RichBar application, which has a RichEdit component you can operate by using the toolbar. The program has buttons for loading and saving files, for copy and paste operations, and to change some of the attributes of the current font.

I won't cover the many details of the RichEdit control's features, which I briefly discussed in a previous chapter, nor discuss the details of this application, which has quite a lot of code. All I'll do is focus on features specific to the ToolBar used by the example and visible in Figure 6.5. This toolbar has buttons, separators, and even a drop-down menu and two combo boxes (discussed in the next section).

FIGURE 6.5
The RichBar example's toolbar. Notice the drop-down menu.


RichBar - E:\books\md7code\06\RichBar\RichBar File Operations.rtf
Arial
clRed
Small
Medium
Large
1/20

The Rich Example

An introduction to the basic features of the RichBar example, discussed in Chapter 6 of the book "Mastering Delphi 7". Written and copyrighted by Marco Cantù.

This document explains how do you create a simple editor based on the RichEdit control, using Delphi 6. The program has a toolbar and implements a number of features, including a complete scheme for opening and saving the text files, discussed in this document. In fact, we want to be able to ask the user to save any modified file before opening a new one, to avoid losing any changes. Sounds like a professional application, doesn't it?

File Operations

The most complex part of this program is implementing the commands of the File pull-down menu-New, Open, Save, and Save As. In each case, we need to track whether the current file has changed,

The various buttons implement features, including opening and saving text files—the program asks the user to save any modified file before opening a new one, to avoid losing any changes. The file-handling portion of the program is quite complex but it is worth exploring, because many file-based applications will use similar code. More details are available in the file `RichBar File Operations.rtf` with the source code for this example, a file you can open with the RichBar program itself.

Besides file operations, the program supports copy and paste operations and font management. The copy and paste operations don't require an interaction with the VCL `Clipboard` object, because the component can handle them with simple commands like these:

```
RichEdit.CutToClipboard;
RichEdit.CopyToClipboard;
RichEdit.PasteFromClipboard;
RichEdit.Undo;
```

It is a little more advanced to know when these operations (and the corresponding buttons) should be enabled. You can enable Copy and Cut buttons when some text is selected, in the `OnSelectionChange` event of the RichEdit control:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
```

```
  tbtnCut.Enabled := RichEdit.SelLength > 0;
  tbtnCopy.Enabled := tbtnCut.Enabled;
end;
```

The Copy operation cannot be determined by an action of the user, because it depends on the content of the Clipboard, which is also influenced by other applications. One approach is to use a timer and check the Clipboard content from time to time. A better approach is to use the `OnIdle` event of the `Application` object (or the ApplicationEvents component). Because the RichEdit control supports multiple Clipboard formats, the code cannot simply look at those, but should ask the component itself, using a low-level feature not surfaced by the Delphi control:

```
procedure TFormRichNote.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
begin
  // update toolbar buttons
  tbtnPaste.Enabled := RichEdit.Perform (em_CanPaste, 0, 0) <> 0;
end;
```

Basic font management is given by the Bold and Italic buttons, which have similar code. The Bold button toggles the relative attribute from the selected text (or changes the style at the current edit position):

```
procedure TFormRichNote.BoldExecute(Sender: TObject);
begin
  with RichEdit.SelAttributes do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
end;
```

Again, the current status of the button is determined by the current selection, so you'll need to add the following line to the `RichEditSelectionChange` method:

```
tbtnBold.Down := fsBold in RichEdit.SelAttributes.Style;
```

A Menu and a Combo Box in a Toolbar

Besides a series of buttons, the RichBar example has a drop-down menu and a couple of combo boxes, a feature shared by many common applications. The drop-down button allows selection of the font size, and the combo boxes allow rapid selection of the font family and the font color. This second combo is built using a ColorBox control.

The Size button is connected to a PopupMenu component (called `SizeMenu`) using the `DropdownMenu` property. A user can click the button, firing its `OnClick` event as usual, or select the drop-down arrow, open the pop-up menu (see again Figure 6.5), and choose one of its options. This case has three possible font sizes, per the menu definition:

```
object SizeMenu: TPopupMenu
  object Small1: TMenuItem
```

```
      Tag = 10
      Caption = 'Small'
      OnClick = SetFontSize
    end
    object Medium1: TMenuItem
      Tag = 16
      Caption = 'Medium'
      OnClick = SetFontSize
    end
    object Large1: TMenuItem
      Tag = 32
      Caption = 'Large'
      OnClick = SetFontSize
    end
  end
```

Each menu item has a tag indicating the actual size of the font, activated by a shared event handler:

```
procedure TFormRichNote.SetFontSize(Sender: TObject);
begin
  RichEdit.SelAttributes.Size := (Sender as TMenuItem).Tag;
end;
```

The ToolBar control is a full-featured control container, so you can take an edit box, a combo box, and other controls and place them directly inside the toolbar. The combo box in the toolbar is initialized in the `FormCreate` method, which extracts the screen fonts available in the system:

```
ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (RichEdit.Font.Name)
```

The combo box initially displays the name of the default font used in the RichEdit control, which is set at design time. This value is recomputed each time the current selection changes, using the font of the selected text, along with the current color for the ColorBox:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ComboFont.ItemIndex :=
    ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name);
  ColorBox1.Selected := RichEdit.SelAttributes.Color;
end;
```

When a new font is selected from the combo box, the reverse action takes place. The text of the current combo box item is assigned as the name of the font for any text selected in the RichEdit control:

```
RichEdit.SelAttributes.Name := ComboFont.Text;
```

The selection of a color in the ColorBox activates similar code.

A Simple Status Bar

Building a status bar is even simpler than building a toolbar. Delphi includes a specific StatusBar component, based on the corresponding Windows common control (a similar control is available in VisualCLX). This component can be used almost like a panel when its `SimplePanel` property is set to True. In this case, you can use the `SimpleText` property to output some text. The real advantage of this component, however, is that it allows you to define a number of subpanels by activating its `Panels` property editor. (You can also display this property editor by double-clicking the status bar control or perform the same operations using the Object TreeView.) Each subpanel has its own graphical attributes, which you can customize using the Object Inspector. Another feature of the status bar component is the "size grip" area added to the lower-right corner of the bar, which is useful for resizing the form. This is a typical element of the Windows user interface, and you can partially control it with the `SizeGrip` property (it auto-disables when the form is not resizable).

A status bar has various uses. The most common is to display information about the menu item currently selected by the user. In addition, a status bar often displays other information about the status of a program: the position of the cursor in a graphical application, the current line of text in a word processor, the status of the lock keys, the time and date, and so on. To show information on a panel, you use its `Text` property, generally in an expression like this:

```
StatusBar1.Panels[1].Text := 'message';
```

In the RichBar example, I've built a status bar with three panels, for command hints, the status of the Caps Lock key, and the current editing position. The StatusBar component of the example actually has four panels—you need to define the fourth in order to delimit the area of the third panel. The last panel is always large enough to cover the remaining surface of the status bar.

TIP *Again, for more detail about the RichBar program, see the RTF file in the example's source code. Notice also that because the hints are to be displayed in the first panel of the status bar, I could have simplified the code by using the* `AutoHint` *property. I preferred showing the more detailed code, so you'll be able to customize it.*

The panels are not independent components, so you cannot access them by name, only by position as in the preceding code snippet. A good solution to improve the readability of a program is to define a constant for each panel you want to use, and then use these constants when referring to the panels. This is my sample code:

```
const
  sbpMessage = 0;
  sbpCaps = 1;
  sbpPosition = 2;
```

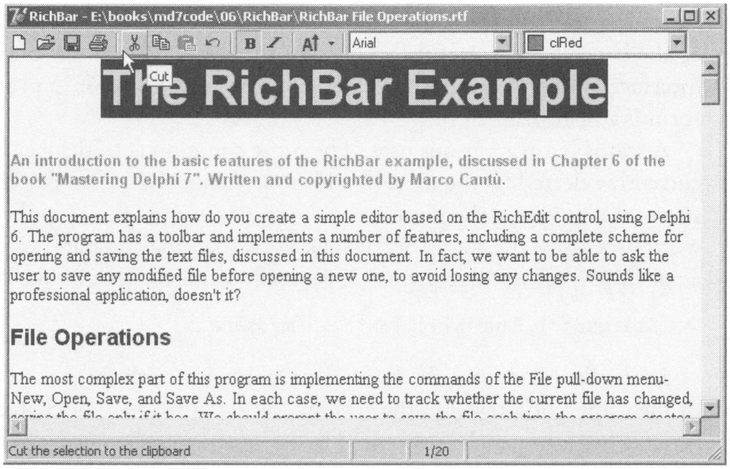
In the first panel of the status bar, I want to display the toolbar button's hint message. The program obtains this effect by handling the application's `OnHint` event, again using the ApplicationEvents component, and copying the current value of the application's `Hint` property to the status bar:

```
procedure TFormRichNote.ApplicationEvents1Hint (Sender: TObject);
begin
  StatusBar1.Panels[sbpMessage].Text := Application.Hint;
end;
```

By default, this code displays in the status bar the same text of the fly-by hints, which aren't generated for menu items. You can use the `Hint` property to specify different strings for the two cases, by writing a string divided into two portions by a separator: the pipe (|) character. For example, you might enter the following as the value of the `Hint` property:

```
'New|Create a new document'
```

The first portion of the string, *New*, is used by fly-by hints, and the second portion, *Create a new document*, by the status bar. You can see an example in Figure 6.6.


RichBar - E:\books\md7code\06\RichBar\RichBar File Operations.rtf
Arial
clRed
Cut

The RichBar Example

An introduction to the basic features of the RichBar example, discussed in Chapter 6 of the book "Mastering Delphi 7". Written and copyrighted by Marco Cantù.

This document explains how do you create a simple editor based on the RichEdit control, using Delphi 6. The program has a toolbar and implements a number of features, including a complete scheme for opening and saving the text files, discussed in this document. In fact, we want to be able to ask the user to save any modified file before opening a new one, to avoid losing any changes. Sounds like a professional application, doesn't it?

File Operations

The most complex part of this program is implementing the commands of the File pull-down menu- New, Open, Save, and Save As. In each case, we need to track whether the current file has changed,

Cut the selection to the clipboard

1/20

FIGURE 6.6
The StatusBar of the RichBar example displays a more detailed description than the fly-by hint.

TIP *When the hint for a control is made up of two strings, you can use the* `GetShortHint` *and* `GetLongHint` *methods to extract the first (short) and second (long) substrings from the string you pass as a parameter, which is usually the value of the* `Hint` *property.*

The second panel displays the status of the Caps Lock key, obtained by calling the `GetKeyState` API function, which returns a state number. If the low-order bit of this number is set (that is, if the number is odd), then the key is pressed. I've decided to check this state when the application is idle, so the test is executed every time a key is pressed but also as soon as a message reaches the window (in case the user changes this setting while working with another program). I've added to the `ApplicationEvents1Idle` handler a call to the custom `CheckCapslock` method, implemented as follows:

```
procedure TFormRichNote.CheckCapslock;
begin
  if Odd (GetKeyState (VK_CAPITAL)) then
    StatusBar1.Panels[sbpCaps].Text := 'CAPS'
  else
    StatusBar1.Panels[sbpCaps].Text := '';
end;
```

Finally, the program uses the third panel to display the current cursor position (measured in lines and characters per line) every time the selection changes. Because the `CaretPos` values are zero-based (that is, the upper-left corner is line 0, character 0), I've added one to each value to make them more reasonable for a casual user:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ...
  // update the position in the status bar
  StatusBar.Panels[sbpPosition].Text := Format ('%d/%d',
    [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```

Themes and Styles

In the past, a GUI-based operating system dictated all the elements of the user interface for programs running on it. In recent years, Linux began to allow users to customize the look-and-feel of both the main windows of applications and user interface controls, like buttons. The same idea (often indicated by the term *skin*) has appeared in numerous programs with such a positive impact that even Microsoft has begun to integrate it (first in programs and then in the entire operating system).

CLX Styles

As I mentioned, on Linux (on XWindow, to be more precise) the user can generally choose the user interface style of the controls. This approach is fully supported by Qt and by the KDE system built on top of it. Qt offers a few basic styles, such as the Windows look-and-feel, the Motif style, and others. A user can also install new styles in the system and make them available to applications.

NOTE *The styles I'm discussing here refer to the user interface of the controls, not of the forms and their borders. This is generally configurable on Linux systems but is technically a separate element of the user interface.*

Because this technique is embedded in Qt, it is also available on the Windows version of the library; CLX makes it available to Delphi developers, so that an application can have a Motif look-and-feel on a Microsoft operating system. The CLX `Application` global object has a `Style` property you can use to set a custom style or a default one, indicated by the `DefaultStyle` subproperty. For example, you can select a Motif look-and-feel with this code:

```
Application.Style.DefaultStyle := dsMotif;
```

In the StylesDemo program, I've added, among various sample controls, a list box with the names of the default styles, as indicated in the `TDefaultStyle` enumeration, and this code for its `OnDblClick` event:

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  Application.Style.DefaultStyle := TDefaultStyle (ListBox1.ItemIndex);
end;
```

The effect is that, by double-clicking the list box, you can change the current application style and immediately see its effect on screen, as demonstrated in Figure 6.7.

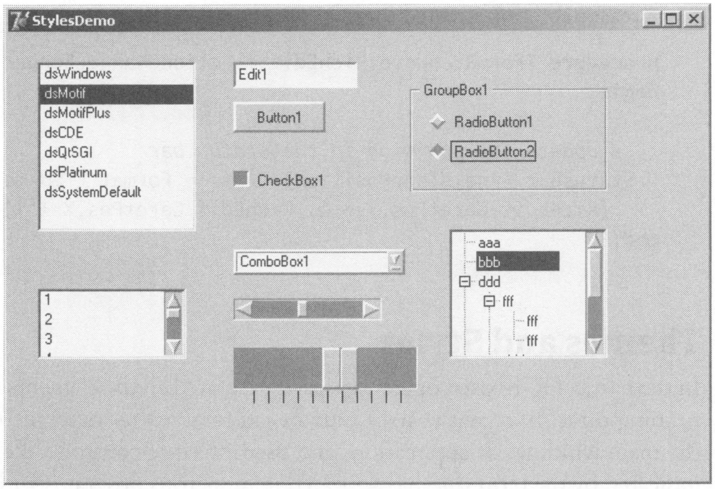


FIGURE 6.7
The StylesDemo program, a Windows application that currently has an unusual Motif layout

Windows XP Themes

With the release of Windows XP, Microsoft has introduced a new, separate version of the common controls library. The old library is still available for compatibility reasons, so that a program running on XP can choose which of the two libraries it wants to use. The new common controls library's main difference is that is doesn't has a fixed rendering engine, but relies on the XP theme engine and delegates the user interface of the controls to the current theme.

In Delphi 7, the VCL fully supports themes, due to a lot of internal code and to the themes management library originally developed by Mike Lischke. Some of these new rendering features are used by the visual controls of the Action Manager architecture, independently of the operating system you are running on. However, full theme support is available only on an operating system that has this feature—at the moment, Windows XP.

Even on XP, Delphi applications use the traditional approach by default. To support XP themes, you must include a manifest file in the program. You can do so multiple ways:

◆ Place a manifest file in the same folder as the application. This is an XML file indicating the identity and the dependencies of the program. The file has the same name as the executable program with an extra `.manifest` extension at the end (as in `MyProgram.exe.manifest`). You can see a sample of such a file in Listing 6.2.

◆ Add the same information in a resource file compiled within the application. You have to write a resource file that includes a manifest file. In Delphi 7, the VCL has a `WindowsXP.res` compiled resource file, which is obtained by recompiling the `WindowsXP.rc` file available among the VCL source files. The resource file includes the `sample.manifest` file, again available among the VCL source files.

- Use the XpManifest component, which Borland has added to Delphi 7 to further simplify this task. As you drop this do-nothing component in a program's form, Delphi will automatically include its XPMan unit, which imports the VCL resource file mentioned earlier.

WARNING *When you remove the XpManifest component from an application, you also have to delete the XPMan unit from the* `uses` *statement manually—Delphi won't do it for you. If you fail to do so, even without the XpManifest component, the program will still bind in the manifest resource file. Using the unit is what really matters (which really makes me wonder why Borland chose to create the component instead of simply providing the unit or the related resource file; by the way the component isn't documented at all).*

LISTING 6.2: A SAMPLE MANIFEST FILE (PAGES.EXE.MANIFEST)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="Pages.exe"
    type="win32"
  />
  <description>Mastering Delphi Demo</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        processorArchitecture="X86"
        publicKeyToken="6595b64144ccf1df"
        language="*"
      />
    </dependentAssembly>
  </dependency>
</assembly>
```

As a demo, I've added the manifest file from Listing 6.2 to the folder of the Pages example discussed at the beginning of this chapter. By running it on Windows XP with the standard XP theme, you'll obtain output similar to that shown in Figure 6.8. You can compare this to Figures 6.1 and 6.2, which display the same program under Windows 2000.

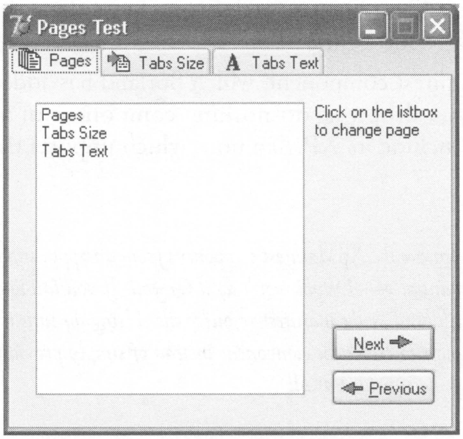


FIGURE 6.8
The Pages example uses the current Windows XP theme, as it includes a manifest file (compare the figure with 6.1)

The ActionList Component

Delphi's event architecture is very open: You can write a single event handler and connect it to the `OnClick` events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, because the event handler can use the `Sender` parameter to refer to the object that fired the event. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, then every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

To overcome this problem, Delphi includes an event-handling architecture based on actions. An *action* (or command) both indicates the operation to do when a menu item or button is clicked and determines the status of all the elements connected to the action. The connection of the action with the user interface of the linked controls is very important and should not be underestimated, because it is where you can get the real advantages of this architecture.

There are many players in this event-handling architecture. The central role is certainly played by the action objects. An action object has a name, like any other component, and other properties that will be applied to the linked controls (called *action clients*). These properties include the `Caption`, the graphical representation (`ImageIndex`), the status (`Checked`, `Enabled`, and `Visible`), and the user feedback (`Hint` and `HelpContext`). There is also the `ShortCut` and a list of `SecondaryShortCuts`, the `AutoCheck` property for two-state actions, the help support properties, and a `Category` property used to arrange actions in logical groups.

The base class for all action objects is `TBasicAction`, which introduces the abstract core behavior of an action, without any specific binding or correction (not even to menu items or controls). The derived `TContainedAction` class introduces properties and methods that enable actions to appear in an action list or action manager. The further-derived `TCustomAction` class introduces support for the properties and methods of menu items and controls that are linked to action objects. Finally, there is the derived ready-to-use `TAction` class.

Each action object is connected to one or more client objects through an ActionLink object. Multiple controls, possibly of different types, can share the same action object, as indicated by their

`Action` property. Technically, the ActionLink objects maintain a bidirectional connection between the client object and the action. The ActionLink object is required because the connection works in both directions. An operation on the object (such as a click) is forwarded to the action object and results in a call to its `OnExecute` event; an update to the status of the action object is reflected in the connected client controls. In other words, one or more client controls can create an ActionLink, which registers itself with the action object.

You should not set the properties of the client controls you connect with an action, because the action will override the property values of the client controls. For this reason, you should generally write the actions first and then create the menu items and buttons you want to connect with them. Note also that when an action has no `OnExecute` handler, the client control is automatically disabled (or grayed), unless the `DisableIfNoHandler` property is set to False.

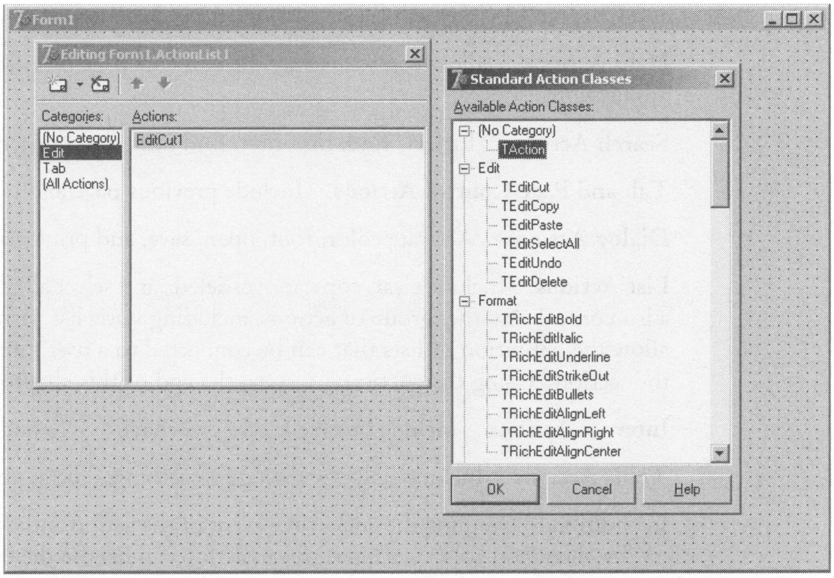
The client controls connected to actions are usually menu items and various types of buttons (pushbuttons, check boxes, radio buttons, speed buttons, toolbar buttons, and the like), but nothing prevents you from creating new components that hook into this architecture. Component writers can even define new actions, as we'll do in Chapter 9 ("Writing Delphi Components"), and new link action objects.

Besides a client control, some actions can also have a target component. Some predefined actions hook to a specific target component. Other actions automatically look for a target component in the form that supports the given action, starting with the active control.

Finally, the action objects are held by an ActionList or ActionManager component, the only class of the basic architecture that shows up on the Component Palette. The action list receives the execute actions that aren't handled by the specific action objects, firing the `OnExecuteAction`. If even the action list doesn't handle the action, Delphi calls the `OnExecuteAction` event of the `Application` object. The ActionList component has a special editor you can use to create several actions, as you can see in Figure 6.9.

FIGURE 6.9

The ActionList component editor, with a list of predefined actions you can use


Form1
Editing Form1.ActionList1
Categories:
Actions:
(No Category)
Edit
Tab
(All Actions)
EditCut1
Standard Action Classes
Available Action Classes:
(No Category)
TAction
Edit
TEditCut
TEditCopy
TEditPaste
TEditSelectAll
TEditUndo
TEditDelete
Format
TRichEditBold
TRichEditItalic
TRichEditUnderline
TRichEditStrikeOut
TRichEditBullets
TRichEditAlignLeft
TRichEditAlignRight
TRichEditAlignCenter
OK
Cancel
Help

In the editor, actions are displayed in groups, as indicated by their `Category` property. By simply setting this property to a new value, you instruct the editor to introduce a new category. These categories are basically logical groups, although in some cases a group of actions can work only on a specific type of target component. You might want to define a category for every pull-down menu or group them in some other logical way.

Predefined Actions in Delphi

With the action list and the ActionManager editor, you can create a new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box, as shown in Figure 6.9. There are many predefined actions, which can be divided into logical groups:

File Actions Include open, save as, open with, run, print setup, and exit.

Edit Actions Illustrated in the next example. They include cut, copy, paste, select all, undo, and delete.

RichEdit Actions Complement the edit actions for RichEdit controls and include bold, italic, underline, strikeout, bullets, and various alignment actions.

MDI Window Actions Demonstrated in Chapter 8, "The Architecture of Delphi Applications," as we examine the Multiple Document Interface approach. They include all the most common MDI operations: arrange, cascade, close, tile (horizontally or vertically), and minimize all.

Dataset Actions Relate to database tables and queries and will be discussed in Chapter 13. There are many dataset actions, representing all the main operations you can perform on a dataset. Delphi 7 adds to the core dataset actions a group of actions specifically tailored to the ClientDataSet component, including apply, revert, and undo. I'll talk more about these actions in Chapter 13 (where I'll cover database programming in general and the ClientDataSet component in particular) and Chapter 14 (in which I'll discuss updating database data).

Help Actions Allow you to activate the contents page or index of the Help file attached to the application.

Search Actions Include find, find first, find next, and replace.

Tab and Page Control Actions Include previous page and next page navigation.

Dialog Actions Activate color, font, open, save, and print dialogs.

List Actions Include clear, copy, move, delete, and select all. These actions let you interact with a list control. Another group of actions, including static list, virtual list, and some support classes, allow the definition of lists that can be connected to a user interface. More on this topic is in the section "Using List Actions" toward the end of this chapter.

Internet Actions Include browse URL, download URL, and send mail actions.

Tools Actions Include only the dialog to customize the action bars.

In addition to handling the `OnExecute` event of the action and changing the status of the action to affect the client controls' user interface, an action can handle the `OnUpdate` event, which is activated when the application is idle. This gives you the opportunity to check the status of the application or the system and change the user interface of the controls accordingly. For example, the standard PasteEdit action enables the client controls only when the Clipboard contains some text.

Actions in Practice

Now that you understand the main ideas behind this important Delphi feature, let's try an example. The program is called Actions, and it demonstrates a number of features of the action architecture. I began building it by placing a new ActionList component in its form and adding the three standard edit actions and a few custom ones. The form also has a panel with some speed buttons, a main menu, and a Memo control (the automatic target of the edit actions). Listing 6.3 is the list of the actions, extracted from the DFM file.

LISTING 6.3: THE ACTIONS OF THE ACTIONS EXAMPLE

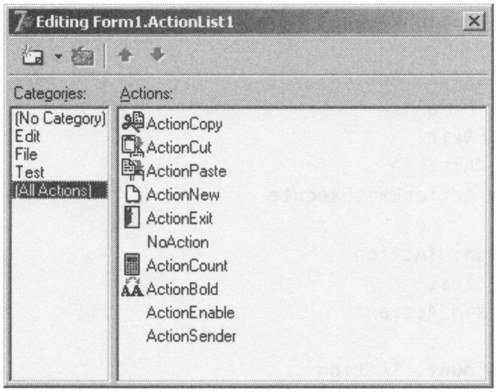
```
object ActionList1: TActionList
  Images = ImageList1
  object ActionCopy: TEditCopy
    Category = 'Edit'
    Caption = '&Copy'
    ShortCut = <Ctrl+C>
  end
  object ActionCut: TEditCut
    Category = 'Edit'
    Caption = 'Cu&t'
    ShortCut = <Ctrl+X>
  end
  object ActionPaste: TEditPaste
    Category = 'Edit'
    Caption = '&Paste'
    ShortCut = <Ctrl+V>
  end
  object ActionNew: TAction
    Category = 'File'
    Caption = '&New'
    ShortCut = <Ctrl+N>
    OnExecute = ActionNewExecute
  end
  object ActionExit: TAction
    Category = 'File'
    Caption = 'E&xit'
    ShortCut = <Alt+F4>
    OnExecute = ActionExitExecute
  end
  object NoAction: TAction
    Category = 'Test'
    Caption = '&No Action'
  end
  object ActionCount: TAction
    Category = 'Test'
```

```
      Caption = '&Count Chars'
      OnExecute = ActionCountExecute
      OnUpdate = ActionCountUpdate
    end
    object ActionBold: TAction
      Category = 'Edit'
      AutoCheck = True
      Caption = '&Bold'
      ShortCut = <Ctrl+B>
      OnExecute = ActionBoldExecute
    end
    object ActionEnable: TAction
      Category = 'Test'
      Caption = '&Enable NoAction'
      OnExecute = ActionEnableExecute
    end
    object ActionSender: TAction
      Category = 'Test'
      Caption = 'Test &Sender'
      OnExecute = ActionSenderExecute
    end
  end
```

NOTE *The shortcut keys are stored in the DFM files using virtual key numbers, which also include values for the Ctrl and Alt keys. In this and other listings throughout the book, I've replaced the numbers with the literal values, enclosing them in angle brackets.*

All these actions are connected to the items of a MainMenu component and some of them also to the buttons of a Toolbar control. Notice that the images selected in the ActionList control affect the actions in the editor only, as you can see in Figure 6.10. In order for the ImageList images to show up in the menu items and in the toolbar buttons, you must also select the image list in the MainMenu and in the Toolbar components.

FIGURE 6.10
The ActionList editor of the Actions example


Editing Form1.ActionList1
Categories:
Actions:
(No Category)
Edit
File
Test
(All Actions)
ActionCopy
ActionCut
ActionPaste
ActionNew
ActionExit
NoAction
ActionCount
ActionBold
ActionEnable
ActionSender

The three predefined actions for the Edit menu don't have associated handlers, but these special objects have internal code to perform the related action on the active edit or memo control. These actions also enable and disable themselves, depending on the content of the Clipboard and on the existence of selected text in the active edit control. Most other actions have custom code, except the `NoAction` object; because it has no code, the menu item and the button connected with this command are disabled, even if the `Enabled` property of the action is set to True.

I've added to the example and to the Test menu another action that enables the menu item connected to the `NoAction` object:

```
procedure TForm1.ActionEnableExecute(Sender: TObject);
begin
  NoAction.DisableIfNoHandler := False;
  NoAction.Enabled := True;
  ActionEnable.Enabled := False;
end;
```

Setting `Enabled` to True produces the effect for only a short time, unless you set the `DisableIfNoHandler` property as discussed in the previous section. Once this operation is done, you disable the current action, because there is no need to issue the same command again.

This is different from an action you can toggle, such as the Edit ➢ Bold menu item and the corresponding speed button. Here is the code of the Bold action (which has the `AutoCheck` property set to True, so that it doesn't need to change the status of the `Checked` property in code):

```
procedure TForm1.ActionBoldExecute(Sender: TObject);
begin
  with Memo1.Font do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
end;
```

The `ActionCount` object has very simple code, but it demonstrates an `OnUpdate` handler; when the memo control is empty, it is automatically disabled. You could obtain the same effect by handling the `OnChange` event of the memo control itself, but in general it might not always be possible or easy to determine the status of a control simply by handling one of its events. Here is the code for the two handlers of this action:

```
procedure TForm1.ActionCountExecute(Sender: TObject);
begin
  ShowMessage ('Characters: ' + IntToStr (Length (Memo1.Text)));
end;

procedure TForm1.ActionCountUpdate(Sender: TObject);
begin
  ActionCount.Enabled := Memo1.Text <> '';
end;
```

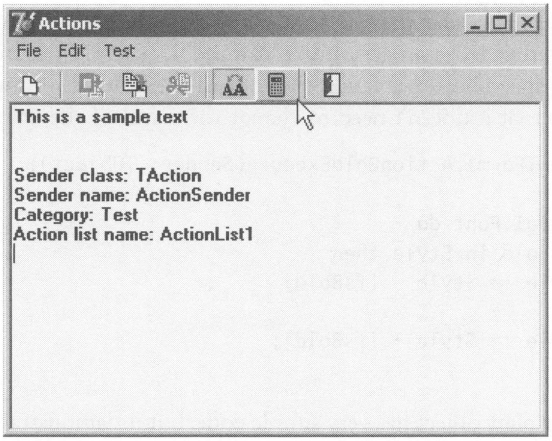
Finally, I've added a special action to test the action event handler's sender object and get some other system information. Besides showing the object class and name, I've added code that accesses the action list object. I've done this mainly to show that you can access this information and how to do it:

```
procedure TForm1.ActionSenderExecute(Sender: TObject);
begin
  Memo1.Lines.Add ('Sender class: ' + Sender.ClassName);
  Memo1.Lines.Add ('Sender name: ' + (Sender as TComponent).Name);
  Memo1.Lines.Add ('Category: ' + (Sender as TAction).Category);
  Memo1.Lines.Add (
    'Action list name: ' + (Sender as TAction).ActionList.Name);
end;
```

You can see the output of this code in Figure 6.11, along with the user interface of the example. Notice that the `Sender` is not the menu item you've selected, even if the event handler is connected to it. The `Sender` object, which fires the event, is the action, which intercepts the user operation.

FIGURE 6.11

The Actions example, with a detailed description of the `Sender` of an Action object's `OnExecute` event



```
This is a sample text


Sender class: TAction
Sender name: ActionSender
Category: Test
Action list name: ActionList1
```

Finally, keep in mind that you can also write handlers for the events of the ActionList object itself, which play the role of global handlers for all the actions in the list, and for the `Application` global object, which fires for all the actions of the application. Before calling the action's `OnExecute` event, Delphi activates the ActionList's `OnExecute` event and the `Application` global object's `OnActionExecute` event. These events can look at the action, eventually execute some shared code, and then stop the execution (using the `Handled` parameter) or let it reach the next level.

If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

NOTE *When an action is executed, it searches for a control to play the role of the action target, by looking at the active control, the active form, and other controls on the form. For example, edit actions refer to the currently active control (if they inherit from* `TCustomEdit`*), and dataset controls look for the dataset connected with the data source of the data-aware control having the input focus. Other actions follow different approaches to find a target component, but the overall idea is shared by most standard actions.*

The Toolbar and ActionList of an Editor

In Chapter 5, I built the RichBar example to demonstrate the development of an editor with a toolbar and a status bar. Of course, I should have also added a menu bar to the form, but doing so would have created quite a few troubles in synchronizing the status of the toolbar buttons with those of the menu items. A very good solution to this problem is to use actions, as I've done in the MdEdit1 example discussed in this section.

The application is based on an ActionList component, which includes actions for file handling and Clipboard support, with code similar to the RichBar version. The font type and color selections are still based on combo boxes, so they don't involve actions—the same is true for the drop-down menu of the Size button. The menu, however, has a few extra commands, including one for character counting and one for changing the background color. These commands are based on actions, and the same is true for the three new paragraph justification buttons (and menu commands).

One of the key differences in this new version is that the code never refers to the status of the toolbar buttons, but eventually modifies the status of the actions. In other cases I've used the actions' `OnUpdate` events. For example, the `RichEditSelectionChange` method doesn't update the status of the Bold button, which is connected to an action with the following `OnUpdate` handler:

```
procedure TFormRichNote.acBoldUpdate(Sender: TObject);
begin
  acBold.Checked := fsBold in RichEdit.SelAttributes.Style;
end;
```

Similar `OnUpdate` event handlers are available for most actions, including the counting operations (available only if there is some text in the RichEdit control), the Save operation (available if the text has been modified), and the Cut and Copy operations (available only if some text is selected):

```
procedure TFormRichNote.acCountcharsUpdate(Sender: TObject);
begin
  acCountChars.Enabled := RichEdit.GetTextLen > 0;
end;

procedure TFormRichNote.acSaveUpdate(Sender: TObject);
begin
  acSave.Enabled := Modified;
end;

procedure TFormRichNote.acCutUpdate(Sender: TObject);
begin
  acCut.Enabled := RichEdit.SelLength > 0;
  acCopy.Enabled := acCut.Enabled;
end;
```

In the older example, the status of the Paste button was updated in the `OnIdle` event of the `Application` object. Now that you're using actions you can convert it into yet another `OnUpdate` handler (see Chapter 5 for details on this code):

```
procedure TFormRichNote.acPasteUpdate(Sender: TObject);
```

```
begin
  acPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;
end;
```

The three paragraph-alignment buttons and the related menu items work like radio buttons: they're mutually exclusive, and one of the three options is always selected. For this reason, the actions have the `GroupIndex` set to 1, the corresponding menu items have the `RadioItem` property set to True, and the three toolbar buttons have their `Grouped` property set to True and the `AllowAllUp` property set to False. (They are also visually enclosed between two separators.)

This arrangement is required so that the program can set the `Checked` property for the action corresponding to the current style, which avoids unchecking the other two actions directly. This code is part of the `OnUpdate` event of the action list, because it applies to multiple actions:

```
procedure TFormRichNote.ActionListUpdate(Action: TBasicAction;
  var Handled: Boolean);
begin
  // check the proper paragraph alignment
  case RichEdit.Paragraph.Alignment of
    taLeftJustify: acLeftAligned.Checked := True;
    taRightJustify: acRightAligned.Checked := True;
    taCenter: acCentered.Checked := True;
  end;
  // checks the caps lock status
  CheckCapslock;
end;
```

Finally, when one of these buttons is selected, the shared event handler uses the value of the `Tag`, set to the corresponding value of the `TAlignment` enumeration, to determine the proper alignment:

```
procedure TFormRichNote.ChangeAlignment(Sender: TObject);
begin
  RichEdit.Paragraph.Alignment := TAlignment ((Sender as TAction).Tag);
end;
```

Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement it differently. Delphi has two ready-to-use toolbar containers:

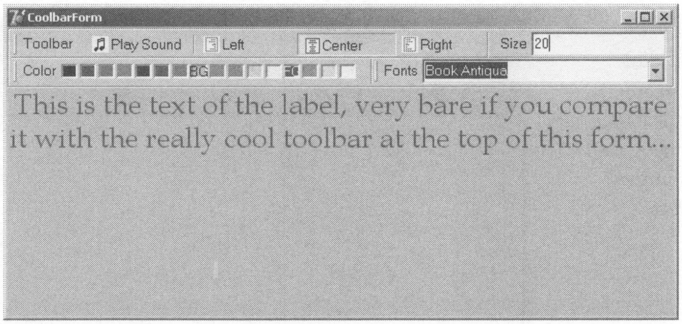
- The CoolBar component is a Win32 common control introduced by Internet Explorer and used by some Microsoft applications.
- The ControlBar component is totally VCL based, with no dependencies on external libraries.

Both components can host toolbar controls as well as some extra elements such as combo boxes and other controls. A toolbar can also replace the menu of an application, as you'll see later. Because the CoolBar component is not frequently used in Delphi applications, it is covered in the sidebar "A Really Cool Toolbar"; the emphasis in the following sections is on Delphi's ControlBar.

A REALLY COOL TOOLBAR

The Win32 CoolBar common control is basically a collection of `TCoolBand` objects that you can activate by using the editor of the Bands property, available also in the component editor menu items or through the Object TreeView. You can customize the CoolBar component in many ways. You can set a bitmap for its background, add some bands to the `Bands` collection, and then assign an existing component or component container to each band. You can use any window-based control (not graphic controls), but only some of them will show up properly. If you want to have a bitmap on the background of the CoolBar, for example, you need to use partially transparent controls. The typical component used in a CoolBar is the Toolbar, but combo boxes, edit boxes, and animation controls are also quite common.

You can place one band on each line or all of them on the same line. Each would use a part of the available surface, and it would be automatically enlarged when the user clicks on its title. It is easier to use this component than to explain it. Try it yourself or open the CoolDemo example:



Size 20

Fonts Book Antiqua

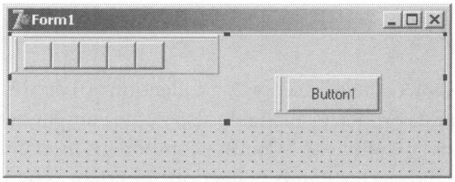
This is the text of the label, very bare if you compare it with the really cool toolbar at the top of this form...

The CoolDemo example's form has a `TCoolBar` component with four bands, two for each of the two lines. The first band includes a subset of the toolbar of the previous example, this time adding an ImageList for the highlighted images. The second has an edit box used to set the font of the text; the third has a ColorGrid component, used to choose the font color and background color. The last band has a ComboBox control with the available fonts.

The user interface of the CoolBar component is nice, and Microsoft uses it in its applications, but alternatives such as the ControlBar component offer a similar UI with no troubles attached. The Windows CoolBar control has had many different and incompatible versions, because Microsoft has released different versions of the common control library with different versions of the Internet Explorer. Some of these versions "broke" existing programs built with Delphi—a very good reason for not using it now even if it is more stable.

The ControlBar

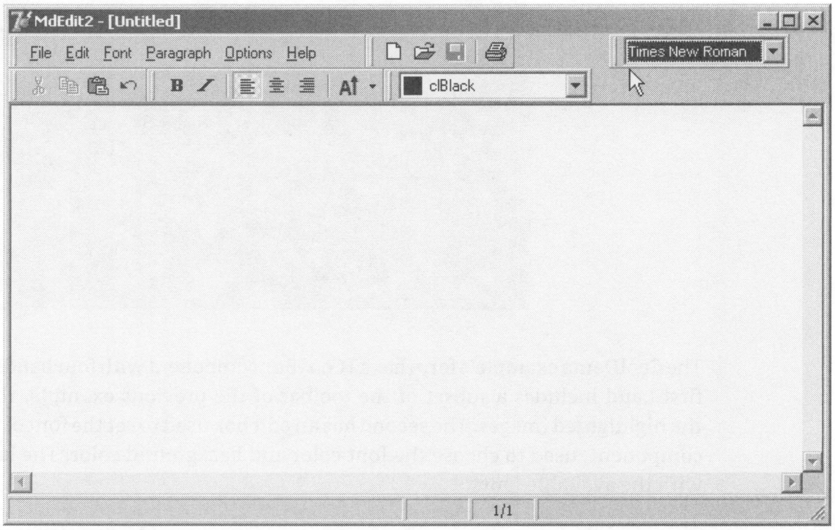
The ControlBar is a control container, and you build it by placing other controls inside it, as you do with a panel (there is no list of `Bands` in it). Every control placed in the bar gets its own dragging area or *grabber* (a small panel with two vertical lines, on the left of the control), even a stand-alone button:



For this reason, you should generally avoid placing specific buttons inside the ControlBar, but instead add containers with buttons inside them. Rather than using a panel, you should use one ToolBar control for every section of the ControlBar.

The MdEdit2 example is another version of the demo I developed to discuss the ActionList component earlier in this chapter. I've grouped the buttons into three toolbars (instead of a single one) and left the two combo boxes as stand-alone controls. All these components are inside a ControlBar so a user can arrange them at runtime, as you can see in Figure 6.12.

FIGURE 6.12
The MdEdit2 example at run time, while a user is rearranging the toolbars in the control bar



The following snippet of the DFM listing of the MdEdit2 example shows how the various toolbars and controls are embedded in the ControlBar component:

```
object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  PopupMenu = BarMenu
  object ToolBarFile: TToolBar
    Flat = True
```

```
      Images = Images
      Wrapable = False
      object ToolButton1: TToolButton
        Action = acNew
      end
      // more buttons...
    end
    object ToolBarEdit: TToolBar...
    object ToolBarFont: TToolBar...
    object ToolBarMenu: TToolBar
      AutoSize = True
      Flat = True
      Menu = MainMenu
    end
    object ComboFont: TComboBox
      Hint = 'Font Family'
      Style = csDropDownList
      OnClick = ComboFontClick
    end
    object ColorBox1: TColorBox...
  end
```

To obtain the standard effect, you have to disable the edges of the toolbar controls and set their style to flat. Sizing all the controls alike, so that you obtain one or two rows of elements of the same height, is not as easy as it might seem at first. Some controls have automatic sizing or various constraints. In particular, to make the combo box the same height as the toolbars, you have to tweak the type and size of its font. Resizing the control itself has no effect.

The ControlBar also has a shortcut menu that allows you to show or hide each of the controls currently inside it. Instead of writing code specific to this example, I've implemented a more generic (and reusable) solution. The shortcut menu, called `BarMenu`, is empty at design time and is populated when the program starts:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
  mItem: TMenuItem;
begin
  ...
  // populate the control bar menu
  for I := 0 to ControlBar.ControlCount - 1 do
  begin
    mItem := TMenuItem.Create (Self);
    mItem.Caption := ControlBar.Controls [I].Name;
    mItem.Tag := Integer (ControlBar.Controls [I]);
    mItem.OnClick := BarMenuClick;
    BarMenu.Items.Add (mItem);
  end;
```

The `BarMenuClick` procedure is a single event handler used by all the menu items; it uses the `Tag` property of the `Sender` menu item to refer to the element of the ControlBar associated with the item in the `FormCreate` method:

```
procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
  aCtrl: TControl;
begin
  aCtrl := TControl ((Sender as TComponent).Tag);
  aCtrl.Visible := not aCtrl.Visible;
end;
```

Finally, the `OnPopup` event of the menu is used to refresh the check mark of the menu items:

```
procedure TFormRichNote.BarMenuPopup(Sender: TObject);
var
  I: Integer;
begin
  // update the menu check marks
  for I := 0 to BarMenu.Items.Count - 1 do
    BarMenu.Items [I].Checked := TControl (BarMenu.Items [I].Tag).Visible;
end;
```

A MENU IN A CONTROL BAR

If you look at the user interface of the MdEdit2 application in Figure 6.12, you'll notice that the form's menu appears inside a toolbar, hosted by the control bar, and below the application caption. All you have to do to accomplish this is set the toolbar's `Menu` property. You must also remove the main menu from the form's `Menu` property (keeping the MainMenu component on the form), to avoid having two copies of the menu on screen.

Delphi's Docking Support

Another feature available in Delphi is support for *dockable* toolbars and controls. In other words, you can create a toolbar and move it to any side of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

Delphi's docking support is connected with container controls, not only with forms. A panel, a ControlBar, and other containers (technically, any control derived from `TWinControl`) can be set up as dock targets by enabling their `DockSite` property. You can also set the `AutoSize` property of these containers, so they'll show up only if they hold a control.

To be able to drag a control (an object of any `TControl`-derived class) into the dock site, simply set its `DragKind` property to dkDock and its `DragMode` property to dmAutomatic. This way, the control can be dragged away from its current position into a new docking container. To undock a component and move it to a special form, you can set its `FloatingDockSiteClass` property to `TCustomDockForm` (to use a predefined stand-alone form with a small caption).

All the docking and undocking operations can be tracked by using special events of the component being dragged (`OnStartDock` and `OnEndDock`) and the component that will receive the docked control (`OnDockOver` and `OnDockDrop`). These docking events are very similar to the dragging events available in earlier versions of Delphi.

There are also commands you can use to accomplish docking operations in code and to explore the status of a docking container. Every control can be moved to a different location using the `Dock`, `ManualDock`, and `ManualFloat` methods. A container has a `DockClientCount` property, indicating the number of docked controls, and a `DockClients` property, which is an array of these controls.

Moreover, if the dock container has the `UseDockManager` property set to True, you'll be able to use the `DockManager` property, which implements the `IDockManager` interface. This interface has many features you can use to customize the behavior of a dock container, including support for streaming its status.

As you can see from this brief description, docking support in Delphi is based on a large number of properties, events, and methods—more features than I have room to explore in detail. The next example introduces the main features you'll need.

NOTE *Docking support is not currently available in VisualCLX on either platform.*

Docking Toolbars in ControlBars

The MdEdit2 example, already discussed, includes docking support. The program has a second ControlBar at the bottom of the form, which accepts dragging one of the toolbars in the ControlBar at the top. Because both toolbar containers have the `AutoSize` property set to True, they are automatically removed when the host contains no controls. I've also set the `AutoDrag` and `AutoDock` properties of both ControlBars to True.

I had to place the bottom ControlBar inside a panel, together with the RichEdit control. Without this trick, the ControlBar, when activated and automatically resized, kept moving below the status bar, which isn't the correct behavior. In the example, the ControlBar is the only panel control aligned to the bottom, so there is no possible confusion.

To let users drag the toolbars out of the original container, you once again (as stated previously) set their `DragKind` property to dkDock and their `DragMode` property to dmAutomatic. The only two exceptions are the menu toolbar, which I decided to keep close to the typical position of a menu bar, and the ColorBox control, because unlike the combo box this component doesn't expose the `DragMode` and `DragKind` properties. (In the example's `FormCreate` method, you'll find code you can use to activate docking for the component, based on the "protected hack" discussed in Chapter 2.) The Fonts combo box can be dragged, but I don't want to let a user dock it in the lower control bar. To implement this constraint, I've used the control bar's `OnDockOver` event handler, by accepting the docking operation only for toolbars:

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept := Source.Control is TToolbar;
end;
```

WARNING *Dragging a toolbar directly from the upper control bar to the lower control bar doesn't work. The control bar doesn't resize to host the toolbar during the dragging operation, as it does when you drag the toolbar to a floating form and then to the lower control bar. This is a bug in the VCL, and it is very difficult to circumvent. As you'll see the next example, MdEdit3 works as expected even if it has the same code: It uses a different component with different VCL support code!*

When you move one of the toolbars outside of any container, Delphi automatically creates a floating form; you might be tempted to set it back by closing the floating form. This doesn't work, because the floating form is removed along with the toolbar it contains. However, you can use the shortcut menu of the topmost ControlBar, also attached to the other ControlBar, to show this hidden toolbar.

The floating form created by Delphi to host undocked controls has a thin caption, the so-called *toolbar caption*, which by default has no text. For this reason, I've added some code to the `OnEndDock` event of each dockable control to set the caption of the newly created form into which the control is docked. To avoid a custom data structure for this information, I've used the text of the `Hint` property for these controls (which is basically not used) to provide a suitable caption:

```
procedure TFormRichNote.EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  if Target is TCustomForm then
    TCustomForm(Target).Caption := GetShortHint((Sender as TControl).Hint);
end;
```

You can see an example of this effect in the MdEdit2 program in Figure 6.13.

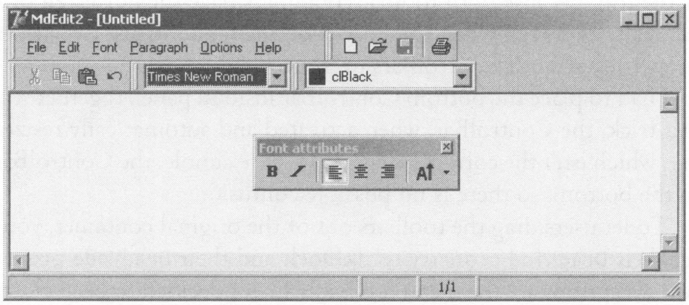


FIGURE 6.13 The MdEdit2 example allows you to dock the toolbars (but not the menu) at the top or bottom of the form or to leave them floating.

Another extension of the example (which I haven't done) might add dock areas on the two sides of the form. The only extra effort this would require would be a routine to turn the toolbars vertically instead of horizontally. Doing so requires switching the `Left` and `Top` properties of each button after disabling the automatic sizing.

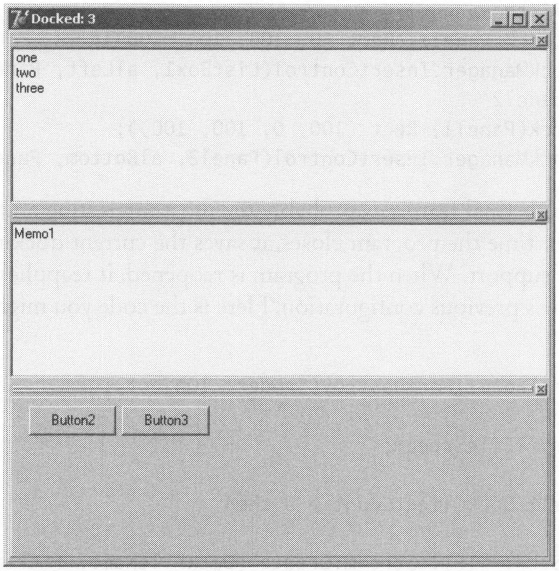
CONTROLLING DOCKING OPERATIONS

Delphi provides many events and methods that give you a lot of control over docking operations, including a dock manager. To explore some of these features, try the DockTest example, a test bed for docking operations shown in Figure 6.14.

The program handles the `OnDockOver` and `OnDockDrop` events of a dock host panel to display messages to the user, such as the number of controls currently docked:

```
procedure TForm1.Panel1DockDrop(Sender: TObject; Source: TDragDockObject;
  X, Y: Integer);
begin
  Caption := 'Docked: ' + IntToStr (Panel1.DockClientCount);
end;
```

FIGURE 6.14
The DockTest example with three controls docked in the main form



In the same way, the program handles the main form's docking events. The controls have a shortcut menu you can invoke to perform docking and undocking operations in code, without the usual mouse dragging, with code like this:

```
procedure TForm1.menuFloatPanelClick(Sender: TObject);
begin
  Panel2.ManualFloat (Rect (100, 100, 200, 300));
end;

procedure TForm1.Floating1Click(Sender: TObject);
var
  aCtrl: TControl;
begin
  aCtrl := Sender as TControl;
  // toggle the floating status
  if aCtrl.Floating then
    aCtrl.ManualDock (Panel1, nil, alBottom)
  else
    aCtrl.ManualFloat (Rect (100, 100, 200, 300));
end;
```

To make the program perform properly at startup, you must dock the controls to the main panel in the initial code; otherwise you can get a weird effect. Oddly enough, for the program to behave properly, you need to add controls to the dock manager and also dock them to the panel (one operation doesn't trigger the other automatically):

```
// dock memo
Memo1.Dock(Panel1, Rect (0, 0, 100, 100));
Panel1.DockManager.InsertControl(Memo1, alTop, Panel1);
```

```
// dock listbox
ListBox1.Dock(Panel1, Rect (0, 100, 100, 100));
Panel1.DockManager.InsertControl(ListBox1, alLeft, Panel1);
// dock panel2
Panel2.Dock(Panel1, Rect (100, 0, 100, 100));
Panel1.DockManager.InsertControl(Panel2, alBottom, Panel1);
```

The example's final feature is probably the most interesting and the most difficult to implement properly. Every time the program closes, it saves the current docking status of the panel, using the dock manager support. When the program is reopened, it reapplies the docking information, restoring the window's previous configuration. Here is the code you might think of writing for saving and loading:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  FileStr: TFileStream;
begin
  if Panel1.DockClientCount > 0 then
  begin
    FileStr := TFileStream.Create (DockFileName, fmCreate or fmOpenWrite);
    try
      Panel1.DockManager.SaveToStream (FileStr);
    finally
      FileStr.Free;
    end;
  end
  else
    // remove the file
    DeleteFile (DockFileName);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  FileStr: TFileStream;
begin
  // initialization code above...

  // reload the settings
  DockFileName := ExtractFilePath (Application.Exename) + 'dock.dck';
  if FileExists (DockFileName) then
  begin
    FileStr := TFileStream.Create (DockFileName, fmOpenRead);
    try
      Panel1.DockManager.LoadFromStream (FileStr);
    finally
      FileStr.Free;
    end;
  end;
  Panel1.DockManager.ResetBounds (True);
end;
```

This code works fine as long as all controls are initially docked. When you save the program, if one control is floating, you won't see it when you reload the settings. However, because of the initialization code inserted earlier, the control will be docked to the panel anyway, and will appear when you drag away the other controls. Needless to say, this is a messy situation. For this reason, after loading the settings, I added this further code:

```
for i := Panel1.DockClientCount - 1 downto 0 do
begin
  aCtrl := Panel1.DockClients[i];
  Panel1.DockManager.GetControlBounds(aCtrl, aRect);
  if (aRect.Bottom - aRect.Top <= 0) then
  begin
    aCtrl.ManualFloat (aCtrl.ClientRect);
    Panel1.DockManager.RemoveControl(aCtrl);
  end;
end;
```

The complete listing includes more commented code, which I used while developing this program; you might use it to understand what happens (which is often different from what you'd expect!). Briefly, the controls that have no size set in the dock manager (the only way I could figure out they are not docked) are shown in a floating window and are removed from the dock manager list.

If you look at the complete code for the `OnCreate` event handler, you'll see a lot of complex code, just to get a plain behavior. You could add more features to a docking program, but to do so you should remove other features, because some of them might conflict. Adding a custom docking form breaks features of the dock manager. Automatic alignments don't work well with the docking manager's code for restoring the status. I suggest you take this program and explore its behavior, extending it to support the type of user interface you prefer.

NOTE *Remember that although docking panels make an application look nice, some users are confused by the fact that their toolbars might disappear or be in a different position than they are used to. Don't overuse the docking features, or some of your inexperienced users may get lost.*

Docking to a PageControl

Another interesting feature of page controls is their specific support for docking. As you dock a new control over a PageControl, a new page is automatically added to host it, as you can easily see in the Delphi environment. To accomplish this, you set the PageControl as a dock host and activate docking for the client controls. This technique works best when you have secondary forms you want to host. Moreover, if you want to be able to move the entire PageControl into a floating window and then dock it back, you'll need a docking panel in the main form.

This is what I've done in the DockPage example, which has a main form with the following settings:

```
object Form1: TForm1
  Caption = 'Docking Pages'
```

```
object Panel1: TPanel
  Align = alLeft
  DockSite = True
  OnMouseDown = Panel1MouseDown
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    DockSite = True
    DragKind = dkDock
    object TabSheet1: TTabSheet
      Caption = 'List'
      object ListBox1: TListBox
        Align = alClient
      end
    end
  end
end
object Splitter1: TSplitter
  Cursor = crHSplit
end
object Memo1: TMemo
  Align = alClient
end
end
```

Notice that the Panel has the `UseDockManager` property set to True and that the PageControl invariably hosts a page with a list box, because when you remove all the pages, the code used for automatic sizing of dock containers might cause you trouble.

The program has two other forms with similar settings (although they host different controls):

```
object Form2: TForm2
  Caption = 'Small Editor'
  DragKind = dkDock
  DragMode = dmAutomatic
  object Memo1: TMemo
    Align = alClient
  end
end
```

You can drag these forms onto the page control to add new pages to it, with captions corresponding with the form titles. You can also undock each of these controls and even the entire PageControl. The program doesn't enable automatic dragging, which would make it impossible to switch pages; instead, the feature is activated when the user clicks on the area of the PageControl that has no tabs—that is, on the underlying panel:

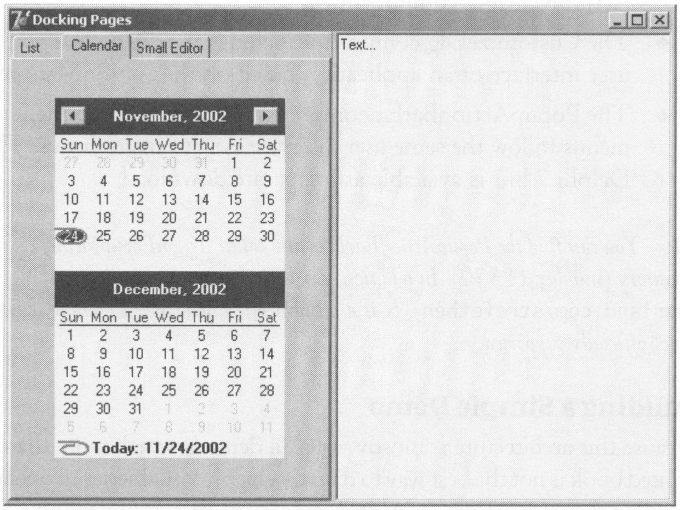
```
procedure TForm1.Panel1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
  PageControl1.BeginDrag (False, 10);
end;
```

You can test this behavior by running the DockPage example, and Figure 6.15 tries to depict it. Notice that when you remove the PageControl from the main form, you cannot directly dock the other forms to the panel, as this is prevented by specific code within the program (simply because at times the behavior won't be correct).

FIGURE 6.15

The main form of the DockPage example after a form has been docked to the page control on the left.



November, 2002

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

December, 2002

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

The ActionManager Architecture

You have seen that actions and the ActionManager component can play a central role in the development of Delphi applications, because they allow a much better separation of the user interface from the actual code of the application. The user interface can now easily change without impacting the code too much. The drawback of this approach is that a programmer has more work to do. To create a new menu item, you need to add the corresponding action first, then move to the menu, add the menu item, and connect it to the action.

To solve this issue, and to provide developers and end users with some advanced features, Delphi 6 introduced a new architecture based on the ActionManager component, which largely extends the role of actions. The ActionManager has a collection of actions as well as a collection of toolbars and menus tied to them. The development of these toolbars and menus is completely visual: You drag actions from a special component editor of the ActionManager to the toolbars to access the buttons you need. Moreover, you can let the end user of your programs do the same operation, rearranging their toolbars and menus beginning with the actions you provide them.

In other words, using this architecture allows you to build applications with a modern user interface, customizable by the user. The menu can show only the recently used items (as many Microsoft programs do), allows for animation, and more.

This architecture is centered on the ActionManager component, but it also includes a few other components found at the end of the Additional page of the palette:

- The ActionManager component is a replacement for the ActionList (but can also use one or more existing ActionLists).
- The ActionMainMenuBar control is a toolbar used to display the menu of an application based on the actions of an ActionManager component.
- The ActionToolBar control is a toolbar used to host buttons based on the actions of an ActionManager component.
- The CustomizeDlg component includes the dialog box you can use to let users customize the user interface of an application based on the ActionManager component.
- The PopupActionBarEx component is an extra component you should use to let your pop-up menus follow the same user interface as your main menus. This component doesn't ship with Delphi 7 but is available as a separate download.

TIP *You can find the PopupActionBarEx (also called ActionPopupMenu) component on Borland's CodeCentral web repository (number 18870). In addition, you'll find more information at the component author's website (*`homepages.borland.com/strefethen`*); he is a member of Delphi's R&D Team at Borland; the component is on the site, but is not officially supported.*

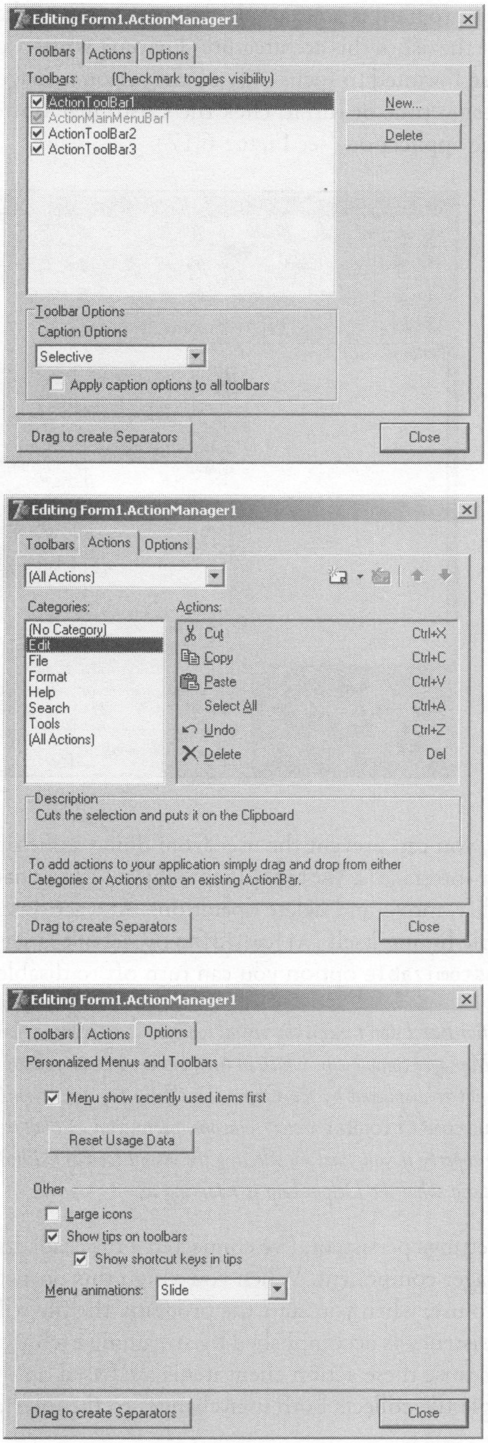
Building a Simple Demo

Because this architecture is mostly visual, a demo is worth more than a general discussion (although a printed book is not the best way to discuss a highly visual series of operations). To create a sample program based on this architecture, drop an ActionManager component on a form and double-click it to open its component editor, shown in Figure 6.16. Notice that this editor is not modal, so you can keep it open while doing other operations in Delphi. This same dialog box is also displayed by the CustomizeDlg component, although with some limited features (for example, adding new actions is disabled).

The editor's three pages are as follows:

- The first page provides a list of visual containers of actions (toolbars or menus). You add new toolbars by clicking the New button. To add new menus, you have to add the corresponding component to the form, then open the `ActionBars` collection of the ActionManager, select an action bar or add a new one, and hook the menu to it using the `ActionBar` property. These are the same steps you could follow to connect a new toolbar to this architecture at run time.
- The second page of the ActionManager editor is very similar to the ActionList editor, providing a way to add new standard or custom actions, arrange them in categories, and change their order. A nice feature of this page, though, is that you can drag a category or a single action from it and drop it onto an action bar control. If you drag a category to a menu, you obtain a pull-down menu with all the category items; if you drag it to a toolbar, each of the category's actions gets a button on the toolbar. If you drag a single action to a toolbar, you get the corresponding button; if you drag it to the menu, you get a direct menu command, which is something you should generally avoid.
- The last page of the ActionManager editor allows you (and optionally an end user) to activate the display of recently used menu items and to modify some of the toolbars' visual properties.

FIGURE 6.16
The three pages of the ActionManager editor dialog box


Editing Form1.ActionManager1
Toolbars
Actions
Options
New...
Delete
Drag to create Separators
Close

Toolbars: (Checkmark toggles visibility)

☑ ActionToolBar1
☑ ActionMainMenuBar1
☑ ActionToolBar2
☑ ActionToolBar3

Toolbar Options

Caption Options: Selective

☐ Apply caption options to all toolbars

(All Actions)

Categories:	Actions:	
(No Category)	Cut	Ctrl+X
Edit	Copy	Ctrl+C
File	Paste	Ctrl+V
Format	Select All	Ctrl+A
Help	Undo	Ctrl+Z
Search	Delete	Del
Tools		
(All Actions)		

Description: Cuts the selection and puts it on the Clipboard

To add actions to your application simply drag and drop from either Categories or Actions onto an existing ActionBar.

Personalized Menus and Toolbars

☑ Menu show recently used items first

Reset Usage Data

Other

☐ Large icons

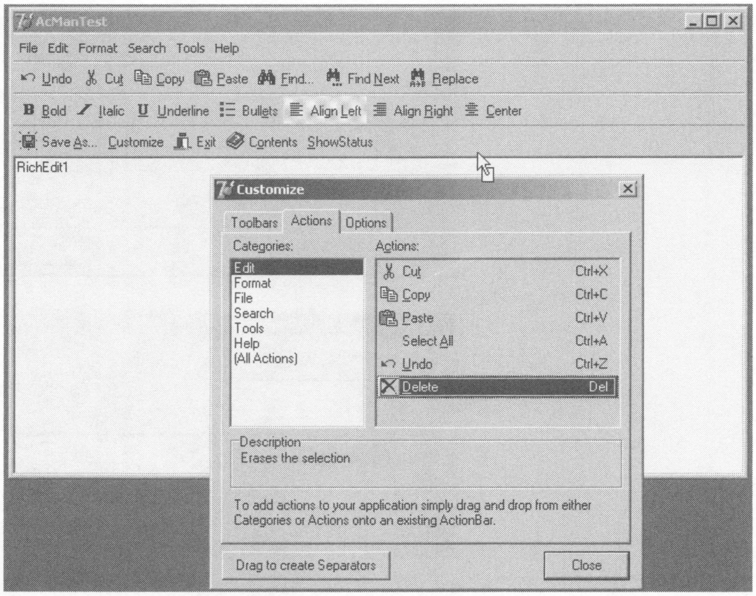
☑ Show tips on toolbars

☑ Show shortcut keys in tips

Menu animations: Slide

The AcManTest program is an example that uses some of the standard actions and a RichEdit control to showcase the use of this architecture (I haven't written any custom code to make the actions work better, because I wanted to focus only on the action manager for this example). You can experiment with it at design time or run it, click the Customize button, and see what an end user can do to customize the application (see Figure 6.17).

FIGURE 6.17
Using the CustomizeDlg component, you can let a user customize the toolbars and the menu of an application by dragging items from the dialog box or moving them around in the action bars.


AcManTest
File Edit Format Search Tools Help
Undo Cut Copy Paste Find... Find Next Replace
Bold Italic Underline Bullets Align Left Align Right Center
Save As... Customize Exit Contents ShowStatus
RichEdit1
Customize
Toolbars Actions Options
Categories:
Edit
Format
File
Search
Tools
Help
(All Actions)
Actions:
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Select All Ctrl+A
Undo Ctrl+Z
Delete Del
Description
Drag to create Separators
Close

Erases the selection

To add actions to your application simply drag and drop from either Categories or Actions onto an existing ActionBar.

In the program, you can prevent the user from doing some operations on actions. Any specific element of the user interface (a `TActionClient` object) has a `ChangedAllowed` property that you can use to disable modify, move, and delete operations. Any action client container (the visual bars) has a property to disable hiding itself (`AllowHiding` by default is set to True). Each ActionBar `Items` collection has a `Customizable` option you can turn off to disable all user changes to the entire bar.

TIP *When I say ActionBar I don't mean the visual toolbars containing action items, but the items of the* `ActionBars` *collection of the ActionManager component, which in turn has an* `Items` *collection. The best way to understand this structure is to look at the subtree displayed by the Object TreeView for an ActionManager component. Each* `TActionBar` *collection item has a* `TCustomActionBar` *visual component connected, but not the reverse (so, for example, you cannot reach this* `Customizable` *property if you start by selecting the visual toolbar). Due to the similarity of the two names, it can take a while to understand what the Delphi help is referring to.*

To make user settings persistent, I've connected a file (called `settings`) to the `FileName` property of the ActionManager component. When you assign this property, you should enter the name of the file you want to use; when you start the program, the file will be created for you by the ActionManager. The persistency is accomplished by streaming each ActionClientItem connected with the action manager. Because these action client items are based on the user settings and maintain state information, a single file collects both user changes to the interface and usage data.

Because Delphi stores user setting and status information in a file you provide, you can make your application support multiple users on a single computer. Simply use a file of settings for each of them (under the MyDocuments or MySettings virtual folder) and connect it to the action manager as the program starts (using the current user of the computer or after some custom login). Another possibility is to store these settings over the network, so that even when a user moves to a different computer, the current personal settings will move along with them.

In the program, I've decided to store the settings in a file store in the same folder as the program, assigning the relative path (the filename) to the ActionManager's `FileName` property. The component will fill in the complete filename with the program folder, easily finding the file to load. However, the file includes among its data its own filename, with an absolute path. So, when it is time to save the file, the operation may refer to an older path. This prevents you from copying this program with its settings to a different folder (for example, this is an issue for the AcManTest demo). You can reset the `FileName` property after loading the file. As a further alternative, you could set the filename at runtime, in the form's `OnCreate` event. In this case you also have to force the file to reload, because you are assigning it after the ActionManager component and the ActionBars have already been created and initialized. However, you might want to force the filename after loading it, as just described:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ActionManager1.FileName :=
    ExtractFilePath (Application.ExeName) + 'settings';
  ActionManager1.LoadFromFile(ActionManager1.FileName);
  // reset the settings file name after loading it (relative path)
  ActionManager1.FileName :=
     ExtractFilePath (Application.ExeName) + 'settings';
end;
```

Least-Recently Used Menu Items

Once a file for the user settings is available, the ActionManager will save the user preferences into it and also use it to track the user activity. This is essential to let the system remove menu items that haven't been used for some time, making them available in an extended menu using the same user interface adopted by Microsoft (see Figure 6.18 for an example).

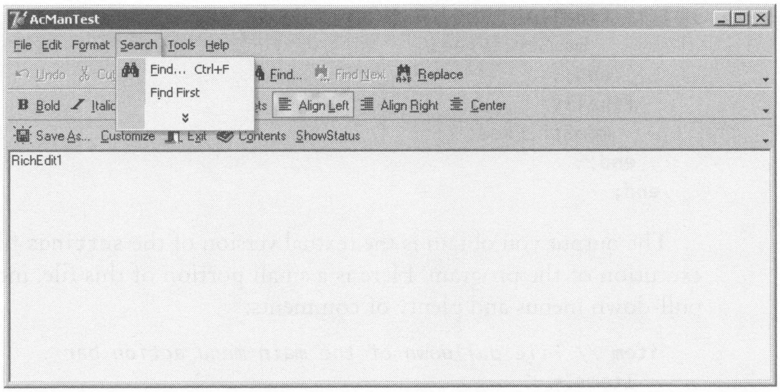


FIGURE 6.18 The ActionManager disables least-recently used menu items that you can still see by selecting the menu extension command.

The ActionManager doesn't just show the least-recently used items: It allows you to customize this behavior in a very precise way. Each action bar has a `SessionCount` property that keeps track of the number of times the application has been executed. Each ActionClientItem has a `LastSession` property and a `UsageCount` property used to track user operations. Notice, by the way, that a user can reset all this dynamic information by using the Reset Usage Data button in the customization dialog.

The system calculates the number of sessions the action has gone unused by computing the difference between the number of times the application has been executed (`SessionCount`) and the last session in which the action has been used (`LastSession`). The value of `UsageCount` is used to look up in the `PrioritySchedule` how many sessions the items can go unused before it is removed. In other words, the `PrioritySchedule` maps each the usage count with a number of *unused* sessions. By modifying the `PrioritySchedule`, you can determine how quickly the items are removed in case they are not used.

You can also prevent this system from being activated for specific actions or groups of actions. The `Items` property of the ActionManager's `ActionBars` has a `HideUnused` property you can toggle to disable this feature for an entire menu. To make a specific item always visible, regardless of the actual usage, you can also set its `UsageCount` property to –1. However, the user settings might override this value.

To help you better understand how this system works, I've added a custom action (`ActionShowStatus`) to the AcManTest example. The action has the following code that saves the current action manager settings to a memory stream, converts the stream to text, and shows it in the memo (refer to Chapter 4 for more information about streaming):

```
procedure TForm1.ActionShowStatusExecute(Sender: TObject);
var
  memStr, memStr2: TMemoryStream;
begin
  memStr := TMemoryStream.Create;
  try
    memStr2 := TMemoryStream.Create;
    try
      ActionManager1.SaveToStream(memStr);
      memStr.Position := 0;
      ObjectBinaryToText(memStr, memStr2);
      memStr2.Position := 0;
      RichEdit1.Lines.LoadFromStream(memStr2);
    finally
      memStr2.Free;
    end;
  finally
    memStr.Free;
  end;
end;
```

The output you obtain is the textual version of the `settings` file automatically updated at each execution of the program. Here is a small portion of this file, including the details of one of the pull-down menus and plenty of comments:

```
item // File pulldown of the main menu action bar
  Items = <
```

```
      item
        Action = Form1.FileOpen1
        LastSession = 19 // was used in the last session
        UsageCount = 4 // was used four times
      end
      item
        Action = Form1.FileSaveAs1 // never used
      end
      item
        Action = Form1.FilePrintSetup1
        LastSession = 7 // used some time ago
        UsageCount = 1 // only once
      end
      item
        Action = Form1.FileRun1 // never used
      end
      item
        Action = Form1.FileExit1 // never used
      end>
    Caption = '&File'
    LastSession = 19
    UsageCount = 5 // the sum of the usage count of the items
  end
```

Porting an Existing Program

If this architecture is useful, you'll probably need to redo most of your applications to take advantage of it. However, if you're already using actions (with the ActionList component), this conversion will be much simpler. The ActionManager has its own set of actions but can also use actions from another ActionManager or ActionList. The ActionManager's `LinkedActionLists` property is a collection of other containers of actions (ActionLists or ActionManagers), which can be associated with the current ActionManager. Associating all the various groups of actions is useful because you can let a user customize the entire user interface with a single dialog box.

If you hook external actions and open the ActionManager editor, you'll see in the Actions page a combo box listing the current ActionManager plus the other action containers linked to it. You can choose one of these containers to see its set of actions and change their properties. The All Action option in this combo box allows you to work on all the actions from the various containers at once; however, I've noticed that at startup it is selected but not always *effective*. Reselect it to see all the actions.

As an example of porting an existing application, I've extended the program built throughout this chapter into the MdEdit3 example. This example uses the same action list as the previous version, hooked to an ActionManager that has the extra customize property to let users rearrange the user interface. Unlike the earlier AcManDemo program, the MdEdit3 example uses a ControlBar as a container for the action bars (a menu, three toolbars, and the usual combo boxes) and has full support for dragging them outside the container as floating bars and dropping them into the lower ControlBar.

To accomplish this, I only had to modify the source code slightly to refer to the new classes for the containers (`TCustomActionToolBar` instead of `TToolBar`) in the `ControlBarLowerDockOver` method. I also found that the ActionToolBar component's `OnEndDock` event passes as parameter an empty target when the system creates a floating form to host the control, so I couldn't easily give this form a new custom caption (see the form's `EndDock` method).

Using List Actions

You'll see more examples of the use of this architecture in the chapters devoted to MDI and database programming (Chapter 8 and Chapter 13, for example). For the moment, I want to add an extra example showing how to use a rather complex group of standard actions: the list actions. List actions comprise two different groups. Some of them (such as Move, Copy, Delete, Clear, and Select All) are normal actions that work on list boxes or other lists. The VirtualListAction and StaticListAction, however, define actions providing a list of items that will be displayed in a toolbar as a combo box.

The ListActions demo highlights both groups of list actions; its ActionManager has five of actions displayed on two separate toolbars. This is a summary of the actions (I've omitted the action bars portion of the component's DFM file):

```
object ActionManager1: TActionManager
  ActionBars.SessionCount = 1
  ActionBars = <...>
  object StaticListAction1: TStaticListAction
    Caption = 'Numbers'
    Items.CaseSensitive = False
    Items.SortType = stNone
    Items = <
      item
        Caption = 'one'
      end
      item
        Caption = 'two'
      end
      ...>
    OnItemSelected = ListActionItemSelected
  end
  object VirtualListAction1: TVirtualListAction
    Caption = 'Items'
    OnGetItem = VirtualListAction1GetItem
    OnGetItemCount = VirtualListAction1GetItemCount
    OnItemSelected = ListActionItemSelected
  end
  object ListControlCopySelection1: TListControlCopySelection
    Caption = 'Copy'
    Destination = ListBox2
    ListControl = ListBox1
  end
  object ListControlDeleteSelection1: TListControlDeleteSelection
    Caption = 'Delete'
  end
  object ListControlMoveSelection2: TListControlMoveSelection
    Caption = 'Move'
    Destination = ListBox2
    ListControl = ListBox1
  end
end
```

The program has also two list boxes in its form, which are used as action targets. The Copy and Move actions are tied to these two list boxes by their `ListControl` and `Destination` properties. The Delete action automatically works with the list box having the input focus.

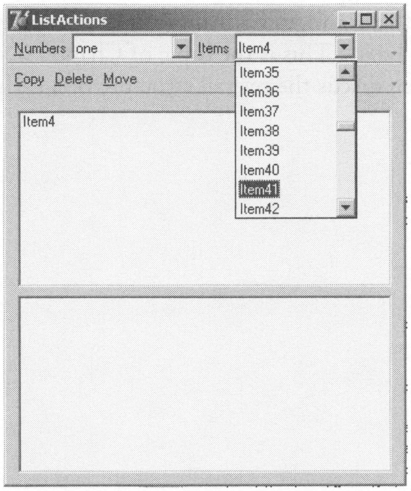
The StaticListAction defines a series of alternative items in its `Items` collection. This is not a plain string list, because any item also has an `ImageIndex` that lets you add graphical elements to the control displaying the list. You can, of course, add more items to this list programmatically. However, if the list is highly dynamic, you can also use the VirtualListAction. This action doesn't define a list of items but has two events you can use to provide strings and images for the list: `OnGetItemCount` allows you to indicate the number of items to display, and `OnGetItem` is then called for each specific item.

In the ListActions demo, the VirtualListAction has the following event handlers for its definition, producing the list you can see in the active combo box in Figure 6.19:

```
procedure TForm1.VirtualListAction1GetItemCount(Sender: TCustomListAction;
  var Count: Integer);
begin
  Count := 100;
end;

procedure TForm1.VirtualListAction1GetItem(Sender: TCustomListAction;
  const Index: Integer; var Value: String;
  var ImageIndex: Integer; var Data: Pointer);
begin
  Value := 'Item' + IntToStr (Index);
end;
```

FIGURE 6.19

The ListActions application has a toolbar hosting a static list and a virtual list.



Numbers: one
Items: Item4
Item4

NOTE *I thought the virtual action items were requested only when needed for display, making this a virtual list. Instead, all the items are created right away. You can prove it by enabling the commented code in the* `VirtualListAction1GetItem` *method (not included in the previous listing), which adds to each item the time its string is requested.*

Both the static list and the virtual list have an `OnItemSelected` event. In the shared event handler, I've written the following code to add the current item to the form's first list box:

```
procedure TForm1.ListActionItemSelected(Sender: TCustomListAction;
  Control: TControl);
begin
  ListBox1.Items.Add ((Control as TCustomActionCombo).SelText);
end;
```

In this case, the sender is the custom action list, but the `ItemIndex` property of this list is not updated with the selected item. However, by accessing the visual control that displays the list, you can obtain the value of the selected item.

What's Next?

In this chapter, I've introduced the use of actions, the actions list, and Action Manager architectures. As you've seen, this is an extremely powerful architecture to separate the user interface from your application code, which uses and refers to the actions and not the menu items or toolbar buttons related to them. The recent extension of this architecture allows users of your programs to have a lot of control, and makes your applications resemble high-end programs without much effort on your part. The same architecture is also very handy for designing your program's user interface, regardless of whether you give this ability to users.

I've also covered some user-interface techniques, such as docking toolbars and other controls. You can consider this chapter the first step toward building professional applications. We will take other steps in the following chapters; but you already know enough to make your programs similar to some best-selling Windows applications, which may be very important for your clients.

Now that the elements of your program's main form are properly set up, you can consider adding secondary forms and dialog boxes. This is the topic of Chapter 7, along with a general introduction to forms. Chapter 8 will then discus the overall structure of a Delphi application.

Chapter 7

Working with Forms

IF YOU'VE READ THE previous chapters, you should now be able to use Delphi's visual components to create the user interface for your applications. Now let's turn our attention to another central element of development in Delphi: forms. You have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the `TForm` class are particularly interesting.

This chapter looks at some of the properties and styles of forms and at sizing and positioning them, as well as form scaling and scrolling. I'll also introduce applications with multiple forms, the use of dialog boxes (custom and predefined ones), frames, and visual form inheritance. Finally, I'll devote some time to input on a form, from both the keyboard and the mouse.

This chapter covers the following topics:

- Form styles, border styles, and border icons
- Mouse and keyboard input
- Direct form painting and special effects
- Positioning, scaling, and scrolling forms
- Creating and closing forms
- Modal and modeless dialog boxes and forms
- Creating secondary forms dynamically
- Predefined dialog boxes
- Building a splash screen

The *TForm* Class

Forms in Delphi are defined by the `TForm` class, which is included in the Forms unit of VCL. Of course, there is now a second definition of forms in VisualCLX. Although I'll mainly refer to the VCL class in this chapter, I'll also highlight differences with the cross-platform version provided in CLX.

The `TForm` class is part of the windowed-controls hierarchy, which starts with the `TWinControl` (or `TWidgetControl`) class. `TForm` inherits from the *almost complete* `TCustomForm`, which in turn inherits from `TScrollingWinControl` (or `TScrollingWidget`). Having all the features of their many base classes, forms have a long series of methods, properties, and events. For this reason, I won't try to list them here—I'd rather present some interesting techniques related to forms throughout this chapter. I'll begin by presenting a technique for *not* defining the form of a program at design time, using the `TForm` class directly, and then explore a few interesting properties of the form class.

Throughout the chapter, I'll point out a few differences between VCL forms and CLX forms. I've built a CLX version for most of the examples, so you can immediately begin experimenting with forms and dialog boxes in CLX, as well as VCL. As in past chapters, the CLX version of each example is prefixed by the letter *Q*.

Using Plain Forms

Delphi developers tend to create forms at design time, which implies deriving a new class from the base class, and build the content of the form visually. This is certainly a reasonable standard practice, but it is not compulsory to create a descendant of the `TForm` class to show a form, particularly if it is a simple one.

Consider this case: Suppose you have to show a rather long message (based on a string) to a user, and you don't want to use the simple predefined message box because it will look too large and won't provide scroll bars. You can create a form with a memo component in it, and display the string inside the memo. Nothing prevents you from creating this form in the standard visual way, but you might consider doing this in code, particularly if you need a large degree of flexibility.

The DynaForm and QDynaForm examples (available among the book source code), which are somewhat extreme, have no form defined at design time but include a unit with this function:

```
procedure ShowStringForm (str: string);
var
  form: TForm;
begin
  Application.CreateForm (TForm, form);
  form.caption := 'DynaForm';
  form.Position := poScreenCenter;
  with TMemo.Create (form) do
  begin
    Parent := form;
    Align := alClient;
    Scrollbars := ssVertical;
    ReadOnly := True;
    Color := form.Color;
    BorderStyle := bsNone;
    WordWrap := True;
    Text := str;
  end;
  form.Show;
end;
```

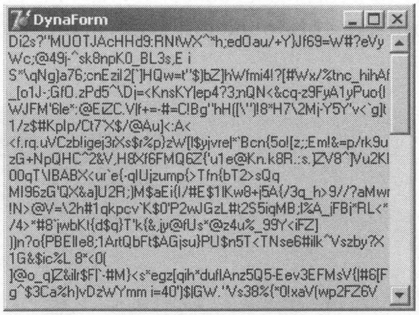
I had to create the form by calling the `Application` global object's `CreateForm` method (a feature required by Delphi applications and discussed in Chapter 8, "The Architecture of Delphi Applications"); other than that, this code does dynamically what you generally do with the Form Designer. Writing this code is undoubtedly more tedious, but it allows also a greater deal of flexibility, because any parameter can depend on external settings.

The previous `ShowStringForm` function is not executed by an event of another form, because there are no traditional forms in this program. Instead, I've modified the project's source code as follows:

```
program DynaForm;

uses
  Forms,
  DynaMemo in 'DynaMemo.pas';

{$R *.RES}

var
  str: string;

begin
  str := '';
  Randomize;
  while Length (str) < 2000 do
    str := str + Char (32 + Random (74));
  ShowStringForm (str);
  Application.Run;
end.
```

The effect of running the DynaForm program is a strange-looking form filled with random characters (as you can see in Figure 7.1)—it isn't terribly useful in itself, but it underscores the idea.

FIGURE 7.1
The dynamic form generated by the DynaForm example is completely created at run time, with no design-time support.



```
Di2s?"MUOTJAcHHd9:RNtWX^"h;ed0au/+Y}Jf69=W#?eVy
Wc;@49j-^sk8npK0_BL3s,E i
S"\qNg}a76;cnEzil2[`]HQw=t"$]bZ]hWfmi4!?[#Wx/%tnc_hihA
_[o1J-;Gf0.zPd5^\Dj=<KnsKYlep4?3;nQN<&cq-z9FyA1yPuo{l
WJFM'6le*:@EiZC.V|f+=-#=C!Bg"hH{[\"]!8*H7\2Mj-Y5Y'v<`g]
1/z$#Kplp/Ct7'X$/@Au]<:A<
<f.rq.uVCzb!igej3rXs$r%p}zW[l$yjvrel*Bcn{5o![z;;Em!&=p/rk9u
zG+NpQHC^2&V,H8Xf6FMQ6Z{'u1e@Kn.k8R.:s.]ZV8^]Vu2Kl
00qT\IBABX<ur`e{-qlUjzump{>Tfn{bT2>sQq
MI96zG'QX&a]U2R;]M$aEi{l/#E$1lKw8+j5A{/3q_h>9//?aMwr
!N>@V=\2h#1qkpcv`K$0'P2wJGzL#t2S5iqMB;I%A_jFBj*RL<*
/4>*#8`jwbKl{d$q}T'k{&,jy@fUs*@z4u%_99Y<iFZ]
]}n?o{PBEIle8;1ArtQbFt$AGjsu}PU$n5T<TNse6#ilk^Vszby?X
1G&$ic%L 8*<0(
]@o_q]Z&ilr$F|`-#M}<s*egz[qih*duflAnz5Q5-Eev3EFMsV{|#6[F
g^$3Ca%h}vDzWYmm i=40']$lGW.'Vs38%{*0!xaV[wp2FZ6V
```

TIP *An indirect advantage of this approach, compared to the use of DFM files for design-time forms, is that it would be much more difficult for an external programmer to grab information about the structure of the application. In Chapter 5, "Visual Controls," you saw that you can extract the DFM from the current Delphi executable file, but the same can be easily accomplished for any executable file compiled with Delphi for which you don't have the source code. If it is important for you to keep to yourself a specific set of components you are using (maybe those in a specific form), along with the default values of their properties, writing the extra code may be worth the effort.*

The Form Style

The `FormStyle` property allows you to choose between a normal form (fsNormal) and the windows that make up a Multiple Document Interface (MDI) application. In this case, you'll use the fsMDIForm style for the MDI parent window—that is, the frame window of the MDI application—and the fsMDIChild style for the MDI child window. To learn more about the development of an MDI application, look at Chapter 8, "The Architecture of Delphi Applications."

A fourth option is the fsStayOnTop style, which determines whether the form must always remain on top of all other windows, except for any that also happen to be "stay-on-top" windows.

To create a top-most form (a form whose window is always on top), you need only set the `FormStyle` property, as indicated earlier. This property has two different effects, depending on the kind of form you apply it to:

- The main form of an application will remain in front of every other application (unless other applications have the same top-most style). At times, this behavior generates a rather ugly visual effect, so it makes sense only for special-purpose alert programs.
- A secondary form will remain in front of any other form in the application it belongs to. The windows of other applications are not affected. This approach is often used for floating toolbars and other forms that should stay in front of the main window.

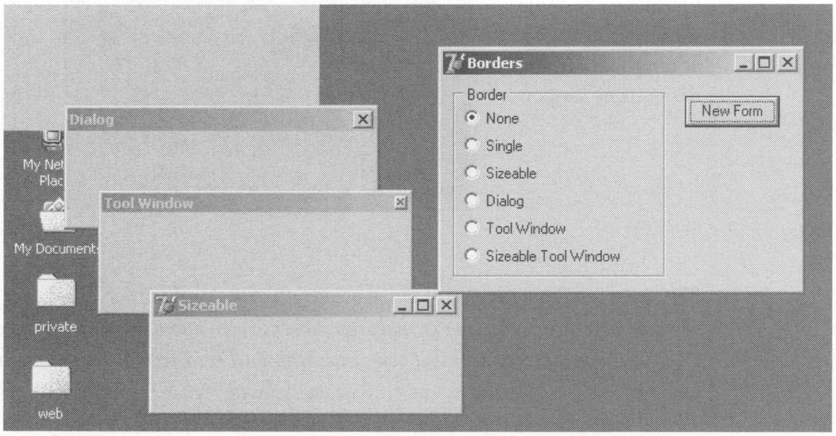
WARNING *In the VCL, when this property is applied to a secondary form, the form only remains in front of the other forms in the same application. In CLX, even a secondary form will be kept in front of any other form of the windowing system—something you'd generally rather avoid.*

The Border Style

Another important property of a form is its `BorderStyle`. This property refers to a visual element of the form, but it has a much more profound influence on the *behavior* of the window, as you can see in Figure 7.2.

FIGURE 7.2

Sample forms with the various border styles, created by the Borders example



Border
● None
○ Single
○ Sizeable
○ Dialog
○ Tool Window
○ Sizeable Tool Window

At design time, the form is always shown using the default value of the `BorderStyle` property, bsSizeable. This value corresponds to a Windows style known as *thick frame.* When a main window has a thick frame around it, a user can resize it by dragging its border. This state is made clear by the special *resize* cursors (with the shape of a double-pointer arrow) displayed when the user moves the mouse onto this thick window border.

A second important choice for this property is bsDialog. If you select it, the form uses as its border the typical dialog-box frame—a thick frame that doesn't allow resizing. In addition to this graphical element, note that if you select the bsDialog value, the form becomes a dialog box. This involves several changes: For example, the items on its system menu are different, and the form will ignore some of the elements of the `BorderIcons` set property.

WARNING *Setting the* `BorderStyle` *property at design time produces no visible effect. Several component properties do not take effect at design time, because they would prevent you from working on the component while developing the program. For example, how could you resize the form with the mouse if it were turned into a dialog box? When you run the application, though, the form will have the border you requested.*

You can assign four other values to the `BorderStyle` property:

- bsSingle creates a main window that's not resizable. Many games and applications based on windows with controls (such as data-entry forms) use this value, simply because resizing these forms makes no sense. Enlarging a form to see an empty area or reducing its size to make some components less visible often doesn't help a program's user (although Delphi's automatic scroll bars partially solve the last problem).
- bsNone is used only in very special situations and inside other forms. You'll never see an application with a main window that has no border or caption (except perhaps as an example in a programming book to show you that it makes no sense).
- bsToolWindow and bsSizeToolWin are related to the specific Win32 extended style `ws_ex_ToolWindow`. This style turns the window into a floating toolbox with a small title font and close button. You should not use this style for the main window of an application.

WARNING *In CLX, the enumeration for the* `BorderStyle` *property uses slightly different values, prefixed by the letters* fbs *(form border style): fbsSingle, fbsDialog, and so on.*

To test the effect and behavior of the different values of the `BorderStyle` property, I've written a program called Borders, available also as QBorders in the CLX version. You've already seen its output in Figure 7.2. However, I suggest you run this example and experiment with it for a while to understand all the differences in the forms. The main form of this program contains only a radio group and a button. The secondary form has no components, and its `Position` property is set to poDefaultPosOnly. This value affects the initial position of the secondary form you'll create by clicking the button. (I'll discuss the `Position` property later in this chapter.)

The program code is simple. When you click the button, a new form is dynamically created, depending on the item selected in the radio group:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);
var
  NewForm: TForm2;
```

```
begin
  NewForm := TForm2.Create (Application);
  NewForm.BorderStyle := TFormBorderStyle (BorderRadioGroup.ItemIndex);
  NewForm.Caption := BorderRadioGroup.Items[BorderRadioGroup.ItemIndex];
  NewForm.Show;
end;
```

This code uses a trick: It casts the number of the selected item into the `TFormBorderStyle` enumeration. This technique works because I've given the radio buttons the same order as the values of the `TFormBorderStyle` enumeration. The `BtnNewFormClick` method then copies the text of the radio button to the caption of the secondary form. This program refers to `TForm2`, the secondary form defined in a secondary unit of the program, which is saved as `Second.pas`. For this reason, to compile the example, you must add the following lines to the `implementation` section of the unit of the main form:

```
uses
  Second;
```

TIP *Whenever you need to refer to another unit of a program, place the corresponding* `uses` *statement in the* `implementation` *portion instead of the* `interface` *portion if possible. Doing so speeds up the compilation process, results in cleaner code (because the units you include are separate from those included by Delphi), and prevents circular unit compilation errors. To refer to other files within the current project, you can also use the File ➢ Use Unit menu command.*

The Border Icons

Another important element of a form is the presence of icons on its border. By default, a window has a small icon connected to the system menu, a Minimize button, a Maximize button, and a Close button on the far right. You can set different options using the `BorderIcons` property, which has four possible values: biSystemMenu, biMinimize, biMaximize, and biHelp.

NOTE *The biHelp border icon enables the "What's this?" Help. When this style is included and the biMinimize and biMaximize styles are excluded, a question mark appears in the form's title bar. If you click this question mark and then click a component inside the form (but not the form itself!), Delphi activates the help for that object (in a pop-up window in Windows 9x, or in a regular WinHelp window in Windows 2000/XP). This behavior is demonstrated by the BIcons example, which has a simple Help file with a page connected to the* `HelpContext` *property of the button in the middle of the form.*

The BIcons example demonstrates the behavior of a form with different border icons and shows how to change this property at run time. The example's form is very simple: It has only a menu, with a pull-down containing four menu items, one for each of the possible elements of the set of border icons. I've written a single method, connected with the four commands, that reads the check marks on the menu items to determine the value of the `BorderIcons` property. This code is therefore also a good exercise in working with sets:

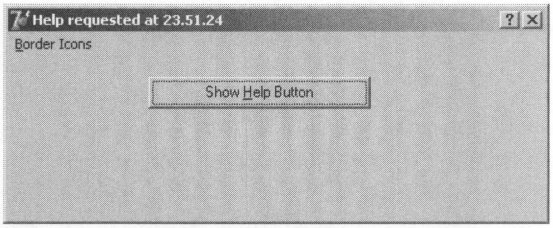
```
procedure TForm1.SetIcons(Sender: TObject);
var
  BorIco: TBorderIcons;
```

```
begin
  (Sender as TMenuItem).Checked := not (Sender as TMenuItem).Checked;
  if SystemMenu1.Checked then
    BorIco := [biSystemMenu]
  else
    BorIco := [];
  if MaximizeBox1.Checked then
    Include (BorIco, biMaximize);
  if MinimizeBox1.Checked then
    Include (BorIco, biMinimize);
  if Help1.Checked then
    Include (BorIco, biHelp);
  BorderIcons := BorIco;
end;
```

While running the BIcons example, you can easily set and remove the various visual elements of the form's border. You'll immediately see that some of these elements are closely related: If you remove the system menu, all the border icons disappear; if you remove either the Minimize or Maximize button, it becomes grayed; if you remove both of these buttons, they disappear. Notice also that in these last two cases, the corresponding items of the system menu are automatically disabled. This is the standard behavior for any Windows application. When the Maximize and Minimize buttons have been disabled, you can activate the Help button. Actually on Windows 2000 if only one of the Maximize and Minimize buttons has been disabled, the Help button will appear but not work. As a shortcut to obtain this effect, you can click the button inside the form. Also, you can click the button after clicking the Help Menu icon to see a help message, as shown in Figure 7.3. As an extra feature, the program also displays in the caption the time the help was invoked, by handling the `OnHelp` event of the form. This effect is visible in the figure.

FIGURE 7.3

The BIcons example. By selecting the Help border icon and clicking the button, you get the help displayed in the figure.


Help requested at 23.51.24

WARNING *If you look at the QBIcons version, built with CLX, you will notice that a bug in the library prevents you from changing the border icons at run time. The different design-time settings work fully, so you'll need to modify the program before running it to see any effect at all. At runtime, the program does nothing!*

Setting More Window Styles

The border style and border icons are indicated by two different Delphi properties, which you can use to set the initial value of the corresponding user interface elements. You have seen that besides changing the user interface, these properties affect the behavior of a window. It is important to know that in VCL (and obviously not in CLX), these border-related properties and the `FormStyle` property primarily correspond to different settings in the *style* and *extended style* of a window. These two terms reflect two parameters of the `CreateWindowEx` API function Delphi uses to create forms.

It is important to acknowledge this fact, because Delphi allows you to modify these two parameters freely by overriding the `CreateParams` virtual method:

```
public
  procedure CreateParams (var Params: TCreateParams); override;
```

This is the only way to use some of the peculiar window styles that are not directly available through form properties. For a list of window styles and extended styles, see the API help under the topics "CreateWindow" and "CreateWindowEx." You'll notice that the Win32 API has styles for these functions, including those related to tool windows.

To show you how to use this approach, I've written the NoTitle example, which lets you create a program with a custom caption. First you must remove the standard caption but keep the resizing frame by setting the corresponding styles:

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := (Params.Style or ws_Popup) and not ws_Caption;
end;
```

To remove the caption, you need to change the overlapped style to a pop-up style; otherwise, the caption will simply stick. To add a custom caption, I've placed a label aligned to the upper border of the form and a small button on the far end. You can see this effect at run time in Figure 7.4.

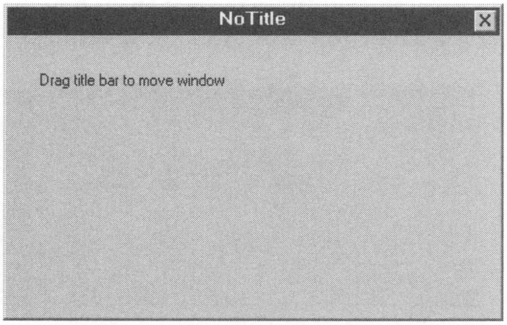


FIGURE 7.4 The NoTitle example has no real caption but a fake one made with a label.

To make the fake caption work, you have to tell the system that a mouse operation on this area corresponds to a mouse operation on the caption. You can do so by intercepting the `wm_NCHitTest` Windows message, which is frequently sent to Windows to determine where the mouse is. When the hit is in the client area and on the label, you can pretend the mouse is on the caption by setting the proper result:

```
procedure TForm1.WMNCHitTest (var Msg: TWMNCHitTest);
  // message wm_NcHitTest
begin
  inherited;
  if (Msg.Result = htClient) and
    (Msg.YPos < Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then
    Msg.Result := htCaption;
end;
```

The `GetSystemMetrics` API function used in this listing queries the operating system about the vertical thickness (`cy`) in pixels of the border around a window with a caption but not sizeable. It is important to make this request every time (and not cache the result), because users can customize most of these elements by using the Appearance page of the Desktop options (in Control Panel) and other Windows settings. The small button has a call to the `Close` method in its `OnClick` event handler. The button is kept in its position even when the window is resized by using the [akTop,akRight] value for the `Anchors` property. The form also has size constraints, so that a user cannot make it too small, as described in the "Form Constraints" section later in this chapter.

Direct Form Input

Having discussed some special capabilities of forms, I'll now move to a very important topic: user input in a form. If you decide to make limited use of components, you might write complex programs as well, receiving input from the mouse and the keyboard. In this chapter, I'll only introduce this topic.

Supervising Keyboard Input

Generally, forms don't handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. In these cases, you can turn on the form's `KeyPreview` property. Then, even if you have some input controls, the form's `OnKeyPress` event will always be activated for any character-input operation (system and shortcut keys excluded). The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character *0*, but the value 0 of the character set, a control character indicated as `#0`).

The example I've built to demonstrate this approach, KPreview, has a form with no special properties (not even `KeyPreview`), a radio group with four options, and some edit boxes, as you can see in Figure 7.5. By default the program does nothing special, except when the various radio buttons are used to enable the key preview:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
  KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```

Now you'll begin receiving the `OnKeyPress` events, and you can do one of the three actions requested by the three special buttons in the radio group. The action depends on the value of the `ItemIndex` property of the radio group component. This is the reason the event handler is based on a `case` statement:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  case RadioPreview.ItemIndex of
    ...
```

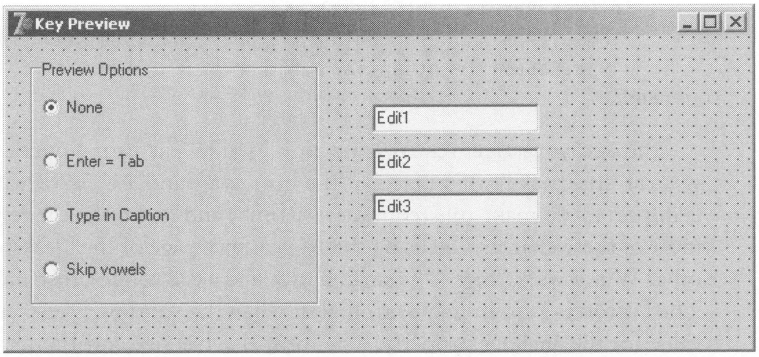


FIGURE 7.5 The KPreview program at design time

In the first case, if the value of the `Key` parameter is `#13`, which corresponds to the Enter key, you disable the operation (setting `Key` to zero) and then mimic the activation of the Tab key. You can do this many ways, but the technique I've chosen is quite particular. I send the `CM_DialogKey` message to the form, passing the code for the Tab key (`VK_TAB`):

```
1: // Enter = Tab
  if Key = #13 then
  begin
    Key := #0;
    Perform (CM_DialogKey, VK_TAB, 0);
  end;
```

NOTE *The* `CM_DialogKey` *message is an internal, undocumented Delphi message. There are a few of them, and it's quite interesting to build advanced components for them and to use them for special coding, but Borland never described them. For more information on this topic, refer to the section "Component Messages and Notifications" in Chapter 9. Notice also that this exact message-based coding style is not available under CLX.*

To type in the form's caption, the program adds the character to the current `Caption`. There are two special cases. When the Backspace key is pressed, the last character of the string is removed (by copying to the `Caption` all the characters of the current `Caption` but the last one). When the Enter key is pressed, the program stops the operation by resetting the `ItemIndex` property of the radio group control. Here is the code:

```
2: // type in caption
begin
  if Key = #8 then // backspace: remove last char
    Caption := Copy (Caption, 1, Length (Caption) - 1)
  else if Key = #13 then // enter: stop operation
    RadioPreview.ItemIndex := 0
  else // anything else: add character
    Caption := Caption + Key;
  Key := #0;
end;
```

Finally, if the last radio item is selected, the code checks whether the character is a vowel (by testing for its inclusion in a constant "vowel set"). In this case, the character is skipped altogether:

```
3: // skip vowels
if UpCase(Key) in ['A', 'E', 'I', 'O', 'U'] then
  Key := #0;
```

Getting Mouse Input

When a user clicks one of the mouse buttons over a form (or over a component), Windows sends the application messages. Delphi defines events you can use to write code that responds to these messages. The two basic events are `OnMouseDown`, received when a mouse button is clicked, and `OnMouseUp`, received when the button is released. Another fundamental system message is related to mouse movement: `OnMouseMove`. Although it should be easy to understand the meaning of the three messages—down, up, and move—you may wonder how they relate to the `OnClick` event you have often used up to now.

You have used the `OnClick` event for components, but it is also available for the form. Its general meaning is that the left mouse button has been clicked and released on the same window or component. However, between these two actions, the cursor might have been moved outside the area of the window or component while the left mouse button was held down.

Another difference between the `OnMouseXX` and `OnClick` events is that the latter relates only to the *left* mouse button. Most of the mouse types connected to a Windows PC have two mouse buttons, and some even have three. Usually you refer to these buttons as the left mouse button (generally used for selection), the right mouse button (for accessing shortcut menus), and the middle mouse button (seldom used).

Nowadays most new mouse devices have a button wheel instead of the middle button; users typically use the wheel for scrolling (causing an `OnMouseWheel` event), but they can also press it (generating the `OnMouseWheelDown` and `OnMouseWheelUp` events). Mouse wheel events are automatically converted into scrolling events.

USING WINDOWS WITHOUT A MOUSE

A user should always be able to use any Windows application without the mouse. This is not an option; it is a Windows programming rule. Of course, an application might be easier to use with a mouse, but that should never be mandatory. Some users may not have a mouse connected, such as travelers with a small laptop and no space, workers in industrial environments, and bank clerks with other peripherals around.

There is another reason to support the keyboard: Using the mouse is nice, but it tends to be slower. If you are a skilled touch typist, you won't use the mouse to drag a word of text; you'll use shortcut keys to copy and paste it without moving your hands from the keyboard.

For these reasons, you should always set up a proper tab order for a form's components. Remember to add keys for buttons and menu items for keyboard selection, use shortcut keys on menu commands, and so on.

THE PARAMETERS OF THE MOUSE EVENTS

All the lower-level mouse events have the same parameters: the usual `Sender` parameter, a `Button` parameter indicating which of the three mouse buttons has been clicked (mbRight, mbLeft, or mbCenter), the `Shift` parameter indicating which of the *mouse-related virtual keys* (the *shift-state modifiers* Alt, Ctrl, and Shift, plus the three mouse buttons) was pressed when the event occurred; and the `x` and `y` coordinates of the position of the mouse in *client area* coordinates of the current window.

Using this information, it is simple to draw a small circle in the position of a left mouse button–down event:

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

NOTE *To draw on the form, you use a special property:* `Canvas`*. A* `TCanvas` *object has two distinctive features: It holds a collection of drawing tools (such as a pen, a brush, and a font) and it has some drawing methods, which use the current tools. The kind of direct drawing code in this example is not correct, because the on-screen image is not persistent; moving another window over the current one will clear its output. The next example demonstrates the Windows "store-and-draw" approach.*

Dragging and Drawing with the Mouse

To demonstrate a few of the mouse techniques discussed so far, I've built an example based on a form without any components. The program is called MouseOne in the VCL version and QMouseOne in the CLX version. It displays the current position of the mouse in the form's `Caption`:

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
end;
```

You can use this feature of the program to better understand how the mouse works. Make this test: Run the program (this simple version or the complete one) and resize the windows on the desktop so that the form of the MouseOne or QMouseOne program is behind another window and inactive but with the title visible. Now move the mouse over the form, and you'll see that the coordinates change. This behavior means the `OnMouseMove` event is sent to the application even if its window is not active, and it proves what I have mentioned: Mouse messages are always directed to the window under the mouse. The only exception is the mouse capture operation I'll discuss in this same example.

Besides showing the position in the title of the window, the MouseOne/QMouseOne example can track mouse movements by painting small pixels on the form if the user keeps the Shift key pressed (again, this direct painting code produces non-persistent output):

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
```

```
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if ssShift in Shift then
    // mark points in yellow
    Canvas.Pixels [X, Y] := clYellow;
end;
```

TIP *The* `TCanvas` *class of the CLX library for Kylix 1 and Delphi 6 didn't include the* `Pixels` *array. Instead, you could call the* `DrawPoint` *method after setting a proper color for the pen, as I've done in the QMouseOne example. Kylix 2 and Delphi 7 re-introduce the* `Pixels` *array property.*

The most interesting feature of this example is its direct mouse-dragging support. Contrary to what you might think, Windows has no system support for dragging, which is implemented in VCL by means of lower-level mouse events and operations. (I discussed an example of dragging from one control to another in Chapter 6.) In VCL, forms cannot originate dragging operations, so in this case you are obliged to use the low-level approach. The aim of this example is to draw a rectangle from the initial position of the dragging operation to the final one, giving users visual clues about the operation they are doing.

The idea behind dragging is quite simple. The program receives a sequence of button-down, mouse-move, and button-up messages. When the button is pressed, dragging begins, although the real actions take place only when the user moves the mouse (without releasing the mouse button) and when dragging terminates (when the button-up message arrives). The problem with this basic approach is that it is not reliable. A window usually receives mouse events only when the mouse is over its client area; so if the user presses the mouse button, moves the mouse onto another window, and then releases the button, the second window will receive the button-up message.

There are two solutions to this problem. One (seldom used) is mouse clipping. Using a Windows API function (`ClipCursor`), you can force the mouse not to leave a certain area of the screen. When you try to move it outside the specified area, it stumbles against an invisible barrier. The second and more common solution is to capture the mouse. When a window captures the mouse, all the subsequent mouse input is sent to that window. This is the approach I've used for the MouseOne/QMouseOne example.

The example's code is built around three methods: `FormMouseDown`, `FormMouseMove`, and `FormMouseUp`. Clicking the left mouse button over the form starts the process, setting the `fDragging` Boolean field of the form (which indicates that dragging is in action in the other two methods). The method also uses a `TRect` variable that keeps track of the initial and current position of the dragging. Here is the code:

```
procedure TMouseForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    fDragging := True;
    Mouse.Capture := Handle;
    fRect.Left := X;
    fRect.Top := Y;
    fRect.BottomRight := fRect.TopLeft;
```

```
      dragStart := fRect.TopLeft;
      Canvas.DrawFocusRect (fRect);
    end;
  end;
```

An important action of this method is the call to the `SetCapture` API function, obtained by setting the `Capture` property of the global object `Mouse`. Now, even if a user moves the mouse outside the client area, the form still receives all mouse-related messages. You can see that behavior by moving the mouse toward the upper-left corner of the screen; the program shows negative coordinates in the caption.

TIP *The global* `Mouse` *object allows you to get global information about the mouse, such as its presence, type, and current position, as well as set some of its global features. This global object hides a few API functions, making your code simpler and more portable. In the VCL the* `Capture` *property has a* `Handle` *type, whereas in CLX it has a* `TControl` *type (the object of the component that captures the mouse). So, the code included in this section will become* `Mouse.Capture := self`*, as you can see in the QMouseOne example.*

When dragging is active and the user moves the mouse, the program draws a dotted rectangle corresponding to the mouse's position. The program calls the `DrawFocusRect` method twice. The first time this method is called, it deletes the current image, thanks to the fact that two consecutive calls to `DrawFocusRect` reset the original situation. After updating the position of the rectangle, the program calls the method a second time:

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
    Canvas.DrawFocusRect (fRect);
    if X > dragStart.X then
      fRect.Right := X
    else
      fRect.Left := X;
    if Y > dragStart.Y then
      fRect.Bottom := Y
    else
      fRect.Top := Y;
    Canvas.DrawFocusRect (fRect);
  end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
end;
```

On Windows 2000 (and other versions) the `DrawFocusRect` function doesn't draw rectangles with a negative size, so the code of the program has been fixed (as you can see above) by comparing the current position with the initial position of the dragging, saved in the `dragStart` point. When the mouse button is released, the program terminates the dragging operation by resetting the `Capture` property of the `Mouse` object (which internally calls the `ReleaseCapture` API function) and by setting the value of the `fDragging` field to False:

```
procedure TMouseForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
  begin
    Mouse.Capture := 0; // calls ReleaseCapture
    fDragging := False;
    Invalidate;
  end;
end;
```

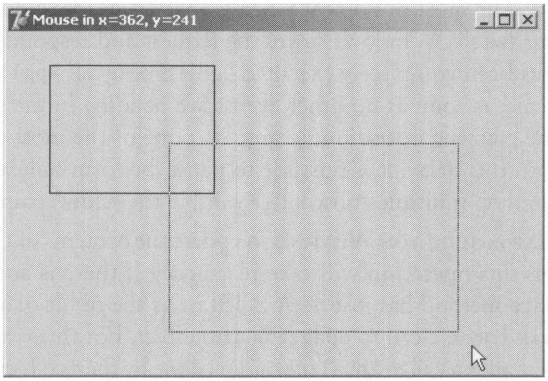
The final call, `Invalidate`, triggers a painting operation and executes the following `OnPaint` event handler:

```
procedure TMouseForm.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle (fRect.Left, fRect.Top, fRect.Right, fRect.Bottom);
end;
```

This makes the output of the form persistent, even if you hide it behind another form. Figure 7.6 shows a previous version of the rectangle and a dragging operation in action.

FIGURE 7.6

During a dragging operation, the MouseOne example uses a dotted line to indicate the final area of a rectangle.



Mouse in x=362, y=241

Painting on Forms

Why do you need to handle the `OnPaint` event to produce proper output, and why can't you paint directly over the form canvas? It depends on Windows' default behavior. As you draw on a window, Windows does *not* store the resulting image. When the window is covered, its contents are usually lost.

The reason for this behavior is simple: to save memory. Windows assumes it's "cheaper" in the long run to redraw the screen using code than to dedicate system memory to preserving the display state of a window. It's a classic memory-versus-CPU-cycles trade-off. A color bitmap for a 600×800 image at 256 colors requires about 480 KB. By increasing the color count or the number of pixels, you can easily reach 4 MB of memory for a 1280×1024 resolution at 16 million colors.

In the event that you want to have consistent output for your applications, you can use two techniques. The general solution is to store enough data about the output to be able to reproduce it when the system sends a *painting* requested. An alternative approach is to save the output of the form in a bitmap while you produce it, by placing an Image component over the form and drawing on the canvas of this image component.

The first technique, painting, is the common approach to handling output in most windowing systems, aside from particular graphics-oriented programs that store the form's whole image in a bitmap. The approach used to implement painting has a very descriptive name: *store and paint.* When the user clicks a mouse button or performs any other operation, you need to store the position and other elements; then, in the painting method, you use this information to paint the corresponding image.

This approach lets the application repaint its whole surface under any of the possible conditions. If you provide a method to redraw the contents of the form, and if this method is automatically called when a portion of the form has been hidden and needs repainting, you will be able to re-create the output properly.

Because this approach takes two steps, you must be able to execute these two operations in a row, asking the system to repaint the window—without waiting for the system to ask for a repaint operation. You can use several methods to invoke repainting: `Invalidate`, `Update`, `Repaint`, and `Refresh`. The first two correspond to the Windows API functions, and the latter two have been introduced by Delphi:

- The `Invalidate` method informs Windows that the entire surface of the form should be repainted. The most important point is that `Invalidate` does *not* enforce a painting operation immediately. Windows stores the request and responds to it only after the current procedure has been completely executed (unless you call `Application.ProcessMessages` or `Update`) and as soon as no other events are pending in the system. Windows deliberately delays the painting operation because it is one of the most time-consuming operations. At times, with this delay, it is possible to paint the form only after multiple changes have taken place, avoiding multiple consecutive calls to the (slow) paint method.
- The `Update` method asks Windows to update the contents of the form, repainting it immediately. However, this operation will take place only if there is an *invalid area.* This happens if the `Invalidate` method has just been called or as the result of an operation by the user. If there is no invalid area, a call to `Update` has no effect. For this reason, it is common to see a call to `Update` just after a call to `Invalidate`, as is done by the two Delphi methods `Repaint` and `Refresh`.
- The `Repaint` method calls `Invalidate` and `Update` in sequence. As a result, it activates the `OnPaint` event immediately. A slightly different version of this method, called `Refresh`, by default calls `Repaint`. The fact that there are two methods for the same operation dates back to Delphi 1 days, when the two were subtly different.

When you need to ask the form for a repaint operation, you should generally call `Invalidate`, following the standard Windows approach. Doing so is particularly important when you need to request this operation frequently, because if Windows takes too much time to update the screen, the requests for repainting can be accumulated into a simple repaint action. The `wm_Paint` message in Windows is a low-priority message; if a request for repainting is pending but other messages are waiting, the other messages are handled before the system performs the paint action.

On the other hand, if you call `Repaint` several times, the screen must be repainted each time before Windows can process other messages; because paint operations are computationally intensive, this behavior can make your application less responsive. Sometimes, however, you want the application to repaint a surface as quickly as possible. In these less-frequent cases, calling `Repaint` is the way to go.

NOTE *Another important consideration is that during a paint operation Windows redraws only the so-called* update region, *to speed up the operation. For this reason, if you invalidate a portion of a window, only that area will be repainted. To accomplish this, you can use the* `InvalidateRect` *and* `InvalidateRegion` *functions. This feature is a double-edged sword: It is a powerful technique that can improve speed and reduce the flickering caused by frequent repaint operations, but, it can also produce incorrect output. A typical problem occurs when only some of the areas affected by the user operations are modified, while others remain as they were even if the system executes the source code that is supposed to update them. If a painting operation falls outside the update region, the system ignores it, as if it were outside the visible area of a window.*

Unusual Techniques: Alpha Blending, Color Key, and the Animate API

One of the recent Delphi features related to forms is support for new Windows APIs that affect the way forms are displayed (in Windows 2000/XP, but not available under Qt/CLX). *Alpha blending* allows you to merge the content of a form with what's behind it on the screen—functionality you'll rarely need, at least in a business application. The technique is more interesting when applied to bitmap (with the new `AlphaBlend` and `AlphaDIBBlend` API functions) than to a form. In any case, by setting the `AlphaBlend` property of a form to True and giving to the `AlphaBlendValue` property a value lower than 255, you'll be able to see in transparency what's behind the form. The lower the `AlphaBlendValue`, the more the form will *fade*. You can see an example of alpha blending in Figure 7.7, taken from the ColorKeyHole example.

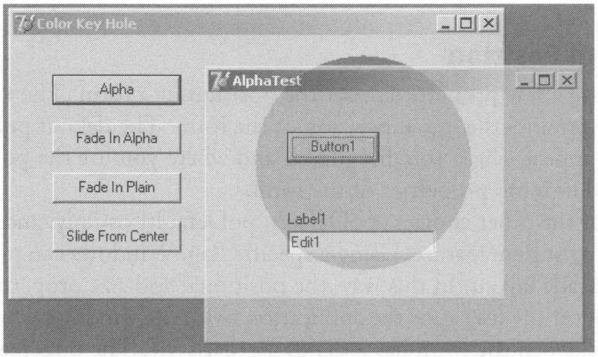


FIGURE 7.7 The output of the ColorKeyHole, showing the effect of the new `TransparentColor` and `AlphaBlend` properties and the AnimateWindow API

Another unusual Delphi feature is the `TransparentColor` boolean property, which allows you to indicate a transparent color that will be replaced by the background, creating a sort of hole in a form. The actual transparent color is indicated by the `TransparentColorValue` property. Again, you can see an example of this effect in Figure 7.7.

Finally, you can use a native Windows technique, *animated display*, which is not directly supported by Delphi (beyond the display of hints). For example, instead of calling the `Show` method of a form, you can write

```
Form3.Hide;
AnimateWindow (Form3.Handle, 2000, AW_BLEND);
Form3.Show;
```

Notice you have to call the `Show` method at the end for the form to behave properly. You can also obtain a similar animation effect by changing the `AlphaBlendValue` property in a loop. The `AnimateWindow` API can also be used to control how the form is brought into view, starting from the center (with the `AW_CENTER` flag) or from one of its sides (`AW_HOR_POSITIVE`, `AW_HOR_NEGATIVE`, `AW_VER_POSITIVE`, or `AW_VER_NEGATIVE`), as is common for slide shows.

You can apply this same function to windowed controls, obtaining a fade-in effect instead of the usual direct appearance. I have serious doubts about the waste of CPU cycles these animations cause, but I must say that if they are applied properly and in the right program, they can improve the user interface.

Position, Size, Scrolling, and Scaling

Once you have designed a form in Delphi, you run the program, and you expect the form to show up exactly as you prepared it. However, a user of your application might have a different screen resolution or might want to resize the form (if this is possible, depending on the border style), eventually affecting the user interface. I've already discussed (mainly in Chapter 7) some techniques related to controls, such as alignment and anchors. Here I'll specifically address elements related to the form as a whole.

Besides differences in the user system, there are many reasons to change Delphi defaults in this area. For example, you might want to run two copies of the program and avoid having all the forms show up in exactly the same place. I've collected many other related elements, including form scrolling, in this portion of the chapter.

The Form Position

You can use a few properties to set the position of a form. The `Position` property indicates how Delphi determines the initial position of the form. The default poDesigned value indicates that the form will appear where you designed it and where you use the positional (`Left` and `Top`) and size (`Width` and `Height`) properties of the form.

Some of the other choices (poDefault, poDefaultPosOnly, and poDefaultSizeOnly) depend on an operating system feature: Using a specific flag, Windows can position and/or size new windows using a cascade layout. In this way, the positional and size properties you set at design time will be ignored, but if the user runs the application twice the windows won't overlap. The default positions are ignored when the form has a dialog border style. The poScreenCenter value displays the form in the center of the screen, with the size you set at design time. This is a common setting for dialog boxes and other secondary forms.

Another property that affects the initial size and position of a window is its *state*. You can use the `WindowState` property at design time to display a maximized or minimized window at startup. This property has only three possible values: wsNormal, wsMinimized, and wsMaximized. If you set a minimized window state, at startup the form will be displayed in the Windows Taskbar. For the main form of an application, this property can be automatically set by specifying the corresponding attributes in a shortcut referring to the application.

Of course, you can maximize or minimize a window at run time, too: Changing the value of the `WindowState` property to wsMaximized or wsNormal produces the expected effect. Setting the property to wsMinimized, however, creates a minimized window that is placed over the Taskbar, not within it. This is not the expected action for a main form, but for a secondary form! The simple solution to this problem is to call the `Minimize` method of the `Application` object. There is also a `Restore` method in the `TApplication` class that you can use when you need to restore a form, although most often the user will do this operation using the system menu's Restore command.

Snapping to the Screen (in Delphi 7)

Forms in Delphi 7 have two new properties:

- The Boolean `ScreenSnap` determines whether the form should be *snapped* to the display area of the screen when it is close to one of its borders.
- The integer `SnapBuffer` determines the distance from the borders considered *close*. Although not a particularly astonishing feature, it's handy to let users snap forms to a side of the screen and take advantage of the entire screen surface; it's particularly handy for applications with multiple forms visible at the same time. Do not set too high a value for the `SnapBuffer` property (something as large as your screen), or the system will become confused!

The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the `Width` and `Height` properties or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the `OnResize` event, where you can perform custom actions to adapt the user interface to the new size of the form).

However, if you look at a form's properties in source code or in the online help, you can see that two properties refer to its width and two refer to its height. `Height` and `Width` refer to the size of the form, including the borders; `ClientHeight` and `ClientWidth` refer to the size of the internal area of the form, excluding the borders, caption, scroll bars (if any), and menu bar. The client area of the form is the surface you can use to place components on the form, to create output, and to receive user input. Notice that in CLX, even `Height` and `Width` refer to the size of the internal area of the form.

Because you may be interested in having a certain available area for your components, it often makes more sense to set the client size of a form instead of its global size. Doing so is straightforward, because as you set one of the two client properties, the corresponding form property changes accordingly.

TIP *In Windows, you can also create output and receive input from the nonclient area of the form—that is, its border. Painting on the border and getting input when you click it are complex issues. If you are interested, look in the Help file at the description of such Windows messages as* `wm_NCPaint`*,* `wm_NCCalcSize`*, and* `wm_NCHitTest`*, and the series of nonclient messages related to the mouse input, such as* `wm_NCLButtonDown`*. The difficulty of this approach is in combining your code with the default Windows behavior.*

Form Constraints

When you choose a resizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the `wm_Size` message, which generates the `OnResize` event. `OnResize` takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

Delphi provides a specific property for forms and also for all controls: the `Constraints` property. Setting the subproperties of the `Constraints` property to the proper maximum and minimum values creates a form that cannot be resized beyond those limits. Here is an example:

```
object Form1: TForm1
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

Notice that as you set up the `Constraints` property, it has an immediate effect even at design time, changing the size of the form if it is outside the permitted area.

Delphi also uses the maximum constraints for maximized windows, producing an awkward effect. For this reason, you should generally disable the Maximize button of a window that has a maximum size. In some cases maximized windows with a limited size make sense—this is the behavior of Delphi's main window. If you need to change constraints at run time, you can also consider using two specific events, `OnCanResize` and `OnConstrainedResize`. The first of the two can also be used to disable resizing a form or control in given circumstances.

Scrolling a Form

When you build a simple application, a single form might hold all the components you need. As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms. If you reduce the space occupied by the components, you might add the capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen (or at least bigger than its visible portion on the screen).

Adding a scroll bar to a form is simple. In fact, you don't need to do anything—if you place several components in a big form and then reduce its size, a scroll bar will be added to the form automatically, as long as you haven't changed the value of the `AutoScroll` property from its default of True.

Along with `AutoScroll`, forms have two properties, `HorzScrollBar` and `VertScrollBar`, which you can use to set several properties of the two `TFormScrollBar` objects associated with the form. The `Visible` property indicates whether the scroll bar is present, the `Position` property determines the initial status of the scroll thumb, and the `Increment` property determines the effect of clicking one of the arrows at the ends of the scroll bar. The most important property, however, is `Range`.

The `Range` property of a scroll bar determines the virtual size of the form, not the range of values of the scroll bar. Suppose you need a form that will host several components and will therefore need to be 1000 pixels wide. You can use this value to set the "virtual range" of the form, changing the `Range` of the horizontal scroll bar.

The `Position` property of the scroll bar will range from 0 to 1000 minus the current size of the client area. For example, if the client area of the form is 300 pixels wide, you can scroll 700 pixels to see the far end of the form (the thousandth pixel).

A SCROLL TESTING EXAMPLE

To demonstrate the specific case I've just discussed, I've built the Scroll1 example, which has a virtual form 1000 pixels wide. I've set the range of the horizontal scroll bar to 1000:

```
object Form1: TForm1
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  OnResize = FormResize
  ...
```

The example's form is filled with meaningless list boxes, and I could have obtained the same scroll-bar range by placing the right-most list box so that its position (`Left`) plus its size (`Width`) equaled 1000.

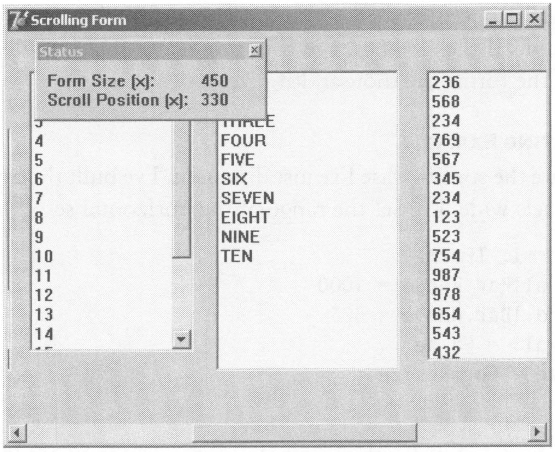
The interesting part of the example is the presence of a toolbox window displaying the status of the form and of its horizontal scroll bar. This second form has four labels: two with fixed text and two with the output. In addition, the secondary form (called `Status`) has a bsToolWindow border style and is a top-most window. You should also set its `Visible` property to True, so its window is automatically displayed at startup:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  FormStyle = fsStayOnTop
  Visible = True
  object Label1: TLabel...
  ...
```

There isn't much code in this program. Its aim is to update the values in the toolbox each time the form is resized or scrolled (as you can see in Figure 7.8). The first part is extremely simple. You can handle the `OnResize` event of the form and copy a couple of values to the two labels. The labels are part of another form, so you need to prefix them with the name of the form instance, `Status`:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

If you wanted to change the output each time the user scrolls the contents of the form, you could not use a Delphi event handler, because forms don't have an `OnScroll` event (although stand-alone ScrollBar components have one). Omitting this event makes sense, because Delphi forms handle scroll bars automatically in a powerful way. In Windows, by contrast, scroll bars are extremely low-level elements, requiring a lot of coding. Handling the scroll event makes sense only in special cases, such as when you want to keep track precisely of the scrolling operations made by a user.



Form Size (x): 450
Scroll Position (x): 330

FIGURE 7.8 The output of the Scroll1 example

Here is the code you need to write. First, add a method declaration to the class and associate it with the Windows horizontal scroll message (`wm_HScroll`); then write the code for this procedure, which is almost the same as the code of the `FormResize` method you've seen before:

```
public
  procedure WMHScroll (var ScrollData: TWMScroll); message wm_HScroll;

procedure TForm1.WMHScroll (var ScrollData: TWMScroll);
begin
  inherited;
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

It's important to add the call to `inherited`, which activates the method related to the same message in the base class form. The `inherited` keyword in Windows message handlers calls the method of the base class you are overriding, which is associated with the corresponding Windows message (even if the procedure name is different). Without this call, the form won't have its default scrolling behavior; that is, it won't scroll at all.

NOTE *Because in CLX you cannot handle the low-level scroll messages, there seems to be no easy way to create a program similar to Scroll1. This isn't terribly important in real-world applications, because the scrolling system is automatic, and you can probably hook in the CLX library at a lower level.*

AUTOMATIC SCROLLING

The scroll bar's `Range` property can seem strange until you begin to use it consistently. When you think about it, you'll start to understand the advantages of the "virtual range" approach. The scroll bar is automatically removed from the form when the client area of the form is big enough to accommodate the virtual size; and when you reduce the size of the form, the scroll bar is added again.

This feature becomes particularly interesting when the `AutoScroll` property of the form is set to True. In this case, the extreme positions of the rightmost and lower controls are automatically copied into the `Range` properties of the form's two scroll bars. Automatic scrolling works well in Delphi. In the previous example, the virtual size of the form would be set to the right border of the last list box. This was defined with the following attributes:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Therefore, the horizontal virtual size of the form would be 977 (the sum of the two preceding values). This number is automatically copied into the `Range` field of the `HorzScrollBar` property of the form, unless you change it manually to have a bigger form (as I've done for the Scroll1 example, setting it to 1000 to leave some space between the last list box and the border of the form). You can see this value in the Object Inspector, or make the following test: Run the program, size the form as you like, and move the scroll thumb to the rightmost position. When you add the size of the form and the position of the thumb, you'll always get 1000, the virtual coordinate of the right-most pixel of the form, whatever the size.

SCROLLING AND FORM COORDINATES

You have just seen that forms can automatically scroll their components. But what happens if you paint directly on the surface of the form? Some problems arise, but their solution is at hand. Suppose you want to draw lines on the virtual surface of a form, as shown in Figure 7.9. Because you probably do not own a monitor capable of displaying 2000 pixels on each axis, you can create a smaller form, add two scroll bars, and set their `Range` property, as I've done in the Scroll2 example.

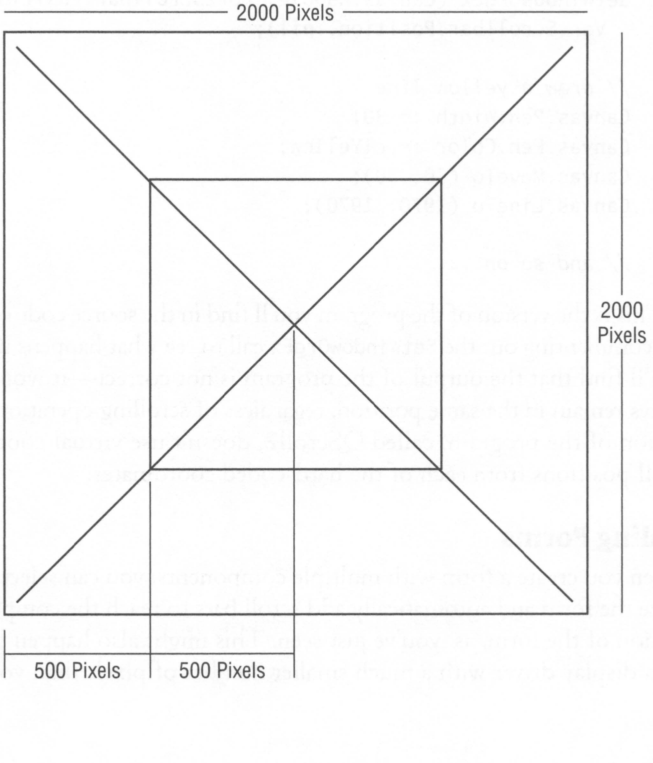


FIGURE 7.9
The lines to draw on the virtual surface of the form

If you draw the lines using the virtual coordinates of the form, the image won't display properly. In the `OnPaint` response method, you need to compute the virtual coordinates yourself. Fortunately, doing so is easy, because you know that the virtual `X1` and `Y1` coordinates of the upper-left corner of the client area correspond to the current positions of the two scroll bars:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  X1, Y1: Integer;
begin
  X1 := HorzScrollBar.Position;
  Y1 := VertScrollBar.Position;

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30-X1, 30-Y1);
  Canvas.LineTo (1970-X1, 1970-Y1);

  // and so on ...
```

As a better alternative, instead of computing the proper coordinate for each output operation, you can call the `SetWindowOrgEx` API to move the origin of the coordinates of the `Canvas`. This way, your drawing code will directly refer to virtual coordinates but will be displayed properly:

```
procedure TForm2.FormPaint(Sender: TObject);
begin
  SetWindowOrgEx (Canvas.Handle, HorzScrollbar.Position,
    VertScrollbar.Position, nil);

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30, 30);
  Canvas.LineTo (1970, 1970);

  // and so on ...
```

This is the version of the program you'll find in the source code of the book. Try using the program and commenting out the `SetWindowOrgEx` call to see what happens if you don't use virtual coordinates: You'll find that the output of the program is not correct—it won't scroll, and the same image will always remain in the same position, regardless of scrolling operations. Notice also that the Qt/CLX version of the program, called QScroll2, doesn't use virtual coordinates but simply subtracts the scroll positions from each of the hard-coded coordinates.

Scaling Forms

When you create a form with multiple components, you can select a fixed-size border or let the user resize the form and automatically add scroll bars to reach the components falling outside the visible portion of the form, as you've just seen. This might also happen because a user of your application has a display driver with a much smaller number of pixels than yours.

Instead of reducing the form size and scrolling the content, you might want to reduce the size of each of the components at the same time. This automatically happens if the user has a system font with a different pixel-per-inch ratio than the one you used for development. To address these problems, Delphi has some nice scaling features, but they aren't fully intuitive.

The form's `ScaleBy` method allows you to scale the form and each of its components. The `PixelsPerInch` and `Scaled` properties let Delphi resize an application automatically when the application is run with a different system font size, often because of a different screen resolution. In both cases, to make the form scale its window, be sure to also set the `AutoScroll` property to False. Otherwise, the contents of the form will be scaled, but the form border itself will not. These two approaches are discussed in the next two sections.

NOTE *Form scaling is calculated based on the difference between the font height at run time and the font height at design time. Scaling ensures that edit and other controls are large enough to display their text using the user's font preferences without clipping the text. The form scales as well, as you will see later, but the main point is to make edit and other controls readable.*

MANUAL FORM SCALING

Any time you want to scale a form, including its components, you can use the `ScaleBy` method, which has two integer parameters, a multiplier and a divisor—it's a fraction. For example, this statement reduces the size of the current form to three-quarters of its original size:

```
ScaleBy (3, 4);
```

The same effect can be obtained by using

```
ScaleBy (75, 100);
```

When you scale a form, all the proportions are maintained, but if you go below or above certain limits, the text strings can alter their proportions slightly. The problem is that in Windows, components can be placed and sized only in whole pixels, whereas scaling almost always involves multiplying by fractional numbers. So, any fractional portion of a component's origin or size will be truncated.

I've built a simple example, Scale (or QScale), to show how you can scale a form manually, responding to a request by the user. The application form has two buttons, a label, an edit box, and an UpDown control connected to it (via its `Associate` property). With this setting, a user can type numbers in the edit box or click the two small arrows to increase or decrease the value (by the amount indicated by the `Increment` property). To extract the input value, you can use the `Text` property of the edit box or the `Position` of the UpDown control. When you click the Do Scale button, the current input value is used to determine the scaling percentage of the form:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);
begin
  AmountScaled := UpDown1.Position;
  ScaleBy (AmountScaled, 100);
  UpDown1.Height := Edit1.Height;
  ScaleButton.Enabled := False;
  RestoreButton.Enabled := True;
end;
```

This method stores the current input value in the form's `AmountScaled` private field and enables the Restore button, disabling the button that was clicked. Later, when the user clicks the Restore button, the opposite scaling takes place. By having to restore the form before another scaling operation takes place, I avoid an accumulation of round-off errors. I've also added a line to set the `Height` of the UpDown component to the same `Height` as the edit box it is attached to. This prevents small differences between the two, due to scaling problems of the UpDown control.

NOTE *If you want to scale the text of the form properly, including the captions of components, the items in list boxes, and so on, you should use TrueType fonts exclusively. The system font (MS Sans Serif) doesn't scale well. The font issue is important because the size of many components depends on the text height of their captions, and if the caption does not scale well, the component might not work properly. For this reason, in the Scale example I've used an Arial font.*

The same scaling technique also works in CLX, as you can see by running the QScale example. The only real difference is that I replaced the UpDown component (and the related edit box) with a SpinEdit control, because the former is not available in Qt.

Automatic Form Scaling

Instead of playing with the `ScaleBy` method, you can have Delphi do the work for you. When Delphi starts, it asks the system for the display configuration and stores the value in the `PixelsPerInch` property of the `Screen` object, a special global object of VCL that's available in any application.

`PixelsPerInch` sounds like it has something to do with the pixel resolution of the screen (actually available in `Screen.Height` and `Screen.Width`), but unfortunately, it doesn't. If you change your screen resolution from 640×480 to 800×600 to 1024×768 or even 1600×1280, you will find that Windows reports the same `PixelsPerInch` value in all cases, unless you change the system font. `PixelsPerInch` really refers to the screen pixel resolution for which the currently installed system font was designed. When a user changes the system font scale, usually to make menus and other text easier to read, the user will expect all applications to honor those settings. An application that does not reflect user desktop preferences will look out of place and, in extreme cases, may be unusable to visually impaired users who rely on very large fonts and high-contrast color schemes.

The most common `PixelsPerInch` values are 96 (small fonts) and 120 (large fonts), but other values are possible. Newer versions of Windows let the user set the system font size to an arbitrary scale. At design time, the `PixelsPerInch` value of the screen, which is a read-only property, is copied to every form of the application. Delphi then uses the value of `PixelsPerInch`, if the `Scaled` property is set to True, to resize the form when the application starts.

As I've mentioned, both automatic scaling and the scaling performed by the `ScaleBy` method operate on components by changing the size of the font. The size of each control depends on the font it uses. With automatic scaling, the value of the form's `PixelsPerInch` property (the design-time value) is compared to the current system value (indicated by the corresponding property of the `Screen` object), and the result is used to change the font of the components on the form. To improve the accuracy of this code, the final height of the text is compared to the design-time height of the text, and its size is adjusted if the heights do not match.

Thanks to Delphi automatic support, the same application running on a system with a different system font size automatically scales itself, without any specific code. The application's edit controls will be the correct size to display their text in the user's preferred font size, and the form will be the correct size to contain those controls. Although automatic scaling has problems in some special cases, if you comply with the following rules, you should get good results:

- Set the `Scaled` property of forms to True (the default value).
- Use only TrueType fonts.
- Use Windows small fonts (96 dpi) on the computer you use to develop the forms.
- Set the `AutoScroll` property to False if you want to scale the form and not just the controls inside it. (`AutoScroll` defaults to True, so don't forget this step.)
- Set the form position either near the upper-left corner or in the center of the screen (with the poScreenCenter value) to avoid having an out-of-screen form.

Creating and Closing Forms

Up to now I have ignored the issue of form creation. You know that when the form is created, you receive the `OnCreate` event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in the project's source file:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box (see Figure 7.10). In this dialog box, you can decide whether the form should be automatically created. If you disable automatic creation, the project's initialization code becomes the following:

```
begin
  Applications.Initialize;
  Application.Run;
end.
```

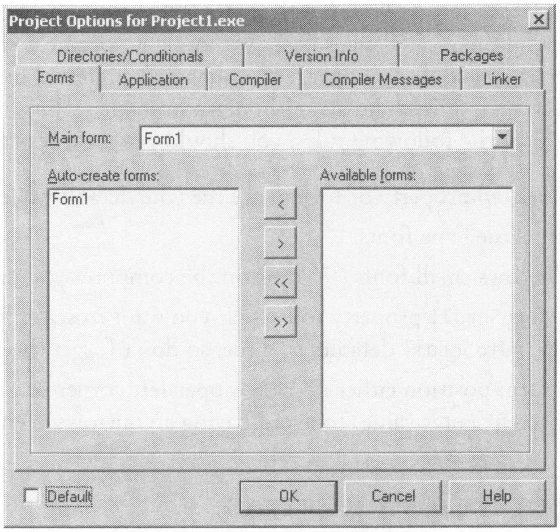
If you now run this program, nothing happens. It terminates immediately because no main window is created. The call to the application's `CreateForm` method creates a new instance of the form class passed as the first parameter and assigns it to the variable passed as the second parameter.

Something else happens behind the scenes. When `CreateForm` is called, if there is currently no main form, the current form is assigned to the application's `MainForm` property. For this reason, the form indicated as Main Form in the dialog box shown in Figure 7.10 corresponds to the first call to the application's `CreateForm` method (that is, when several forms are created at startup).

The same holds for closing the application. Closing the main form terminates the application, regardless of the other forms. If you want to perform this operation from the program's code, call the `Close` method of the main form, as you've done several times in past examples.

FIGURE 7.10

The Forms page of the Delphi Project Options dialog box



Main form: Form1

Auto-create forms:
Form1

Available forms:

☐ Default

Form Creation Events

Regardless of the manual or automatic creation of forms, when a form is created, you can intercept many events. Form-creation events are fired in the following order:

1. `OnCreate` indicates that the form is being created.
2. `OnShow` indicates that the form is being displayed. Besides main forms, this event happens after you set the `Visible` property of the form to True or call the `Show` or `ShowModal` method. This event is fired again if the form is hidden and then displayed again.
3. `OnActivate` indicates that the form becomes the active form within the application. This event is fired every time you move from another form of the application to the current one.
4. Other events, including `OnResize` and `OnPaint`, indicate operations always done at startup but then repeated many times.

NOTE *In Qt, the* `OnResize` *event won't fire as it does in Windows when the form is created. To make the code more portable from Delphi to Kylix, CLX simulates this event, although it would make more sense to tweak the VCL to avoid this odd behavior (a comment in the CLX source code describes this situation).*

As you can see in the previous list, every event has a specific role apart from form initialization, except `OnCreate`, which is guaranteed to be called only once as the form is created.

However, there is an alternative approach to adding initialization code to a form: overriding the constructor. This is usually done as follows:

```
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited Create (AOwner);
  // extra initialization code
end;
```

Before the call to the `Create` method of the base class, the properties of the form are still not loaded and the internal components are not available. For this reason the standard approach is to call the base class constructor first and then do the custom operations.

NOTE *Up to version 3, Delphi used a different creation order, which has led to the* `OldCreateOrder` *compatibility property of the VCL. When this property is set to the default value of False, all the code in a form constructor is executed before the code in the* `OnCreate` *event handler (which is fired by the special* `AfterConstruction` *method). If you enable the old creation order, instead, the constructor's* `inherited` *call leads to the call of the* `OnCreate` *event handler. You can examine the behavior of the CreateOrd example using the two values of the* `OldCreateOrder` *property.*

Closing a Form

When you close the form using the `Close` method or by the usual means (Alt+F4, the system menu, or the Close button), the `OnCloseQuery` event is called. In this event, you can ask the user to confirm the action, particularly if there is unsaved data in the form. Here is an example of the code you can write:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg ('Are you sure you want to exit?', mtConfirmation,
      [mbYes, mbNo], 0) = mrNo then
    CanClose := False;
end;
```

If `OnCloseQuery` indicates that the form should still be closed, the `OnClose` event is called. The third step is to call the `OnDestroy` event, which is the opposite of the `OnCreate` event and is generally used to de-allocate objects related to the form and free the corresponding memory.

NOTE *To be more precise, the* `BeforeDestruction` *method generates an* `OnDestroy` *event before the* `Destroy` *destructor is called. That is, unless you have set the* `OldCreateOrder` *property to True, in which case Delphi uses a different closing sequence.*

So, what is the use of the intermediate `OnClose` event? In this method, you have another chance to avoid closing the application, or you can specify alternative "close actions." The method has an `Action` parameter passed by reference. You can assign the following values to this parameter:

caNone The form is not allowed to close. This corresponds to setting the `CanClose` parameter of the `OnCloseQuery` method to False.

caHide The form is not closed, just hidden. This makes sense if there are other forms in the application; otherwise, the program terminates. This is the default for secondary forms, and it's the reason I had to handle the `OnClose` event in the previous example to close the secondary forms.

caFree The form is closed, freeing its memory, and the application eventually terminates if this was the main form. This is the default action for the main form and the action you should use when you create multiple forms dynamically (if you want to remove the windows and destroy the corresponding Delphi object as the form closes).

caMinimize The form is not closed but only minimized. This is the default action for MDI child forms.

NOTE *When a user shuts down Windows, the* `OnCloseQuery` *event is activated, and a program can use it to stop the shutdown process. In this case, the* `OnClose` *event is not called even if* `OnCloseQuery` *sets the* `CanClose` *parameter to True.*

Dialog Boxes and Other Secondary Forms

When you write a program, there is no significant difference between a dialog box and another secondary form, aside from the border, the border icons, and similar user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have nonmodal—or *modeless*—dialog boxes.

So, if you think dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. You must consider two different elements: The form's border and its user interface determine whether it looks like a dialog box; the use of two different methods (`Show` and `ShowModal`) to display the secondary form determines its behavior (modeless or modal).

Adding a Second Form to a Program

To add a second form to an application, you click the New Form button on the Delphi toolbar or use the File ➢ New ➢ Form menu command. As an alternative, you can select File ➢ New ➢ Other, move to the Forms or Dialogs page, and choose one of the available form templates or form wizards.

If you have two forms in a project, you can use the View Form or View Unit button on the Delphi toolbar to navigate through them at design time. You can also choose which form is the main one and which forms should be automatically created at startup using the Forms page of the Project Options dialog box. This information is reflected in the source code of the project file.

TIP *Secondary forms are automatically created in the project source-code file depending on the status of the Auto Create Forms check box on the Designer page of the Environment Options dialog box. Although automatic creation is the simplest and most reliable approach for novice developers and quick-and-dirty projects, I suggest that you disable this check box for any serious development. When your application contains hundreds of forms, they shouldn't all be created at application startup. Create instances of secondary forms when and where you need them, and free them when you're done.*

Once you have prepared the secondary form, you can set its `Visible` property to True, and both forms will show up as the program starts. In general, the secondary forms of an application are left "invisible" and are then displayed by calling the `Show` method (or setting the `Visible` property at run time). If you use the `Show` function, the second form will be displayed as modeless, so you can move back to the first form while the second is still visible. To close the second form, you might use its system menu or click a button or menu item that calls the `Close` method. As you've just seen, the default close action (see the `OnClose` event) for a secondary form is simply to hide it, so the secondary form is not destroyed when it is closed. It is kept in memory (again, not always the best approach) and is available if you want to show it again.

Creating Secondary Forms at Run Time

Unless you create all the forms when the program starts, you'll need to check whether a form exists and create it if necessary. The simplest case occurs when you want to create multiple copies of the same form at run time. In the MultiWin/QMultiWin example, I've done this by writing the following code:

```
with TForm3.Create (Application) do
  Show;
```

Every time you click the button, a new copy of the form is created. Notice that I don't use the `Form3` global variable, because it doesn't make much sense to assign this variable a new value every time you create a new form object. The important thing, however, is not to refer to the global `Form3` object in the code of the form or in other portions of the application. The `Form3` variable will invariably be a pointer to `nil`. My suggestion, in such a case, is to remove it from the unit to avoid any confusion.

TIP *In the code of a form that can have multiple instances, you should never explicitly refer to the form by using the global variable Delphi sets up for it. For example, suppose that in the code for* `TForm3` *you refer to* `Form3.Caption`*. If you create a second object of the same type (the class* `TForm3`*), the expression* `Form3.Caption` *will refer to the caption of the form object referenced by the* `Form3` *variable, which might not be the current object executing the code. To avoid this problem, refer to the* `Caption` *property in the form's method to indicate the caption of the current form object, and use the* `Self` *keyword when you need a specific reference to the object of the current form. To avoid any problem when creating multiple copies of a form, I suggest removing the global form object from the interface portion of the unit declaring the form. This global variable is required only for the automatic form creation.*

When you create multiple copies of a form dynamically, remember to destroy each form object as is it closed, by handling the corresponding event:

```
procedure TForm3.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Failing to do so will result in a lot of memory consumption, because all the forms you create (both the windows and the Delphi objects) will be kept in memory and hidden from view.

CREATING SINGLE-INSTANCE SECONDARY FORMS

Now let's focus on the dynamic creation of a form, in a program that accounts for only one copy of the form at a time. Creating a modal form is quite simple, because the dialog box can be destroyed when it is closed, with code like this:

```
var
  Modal: TForm4;
begin
  Modal := TForm4.Create (Application);
  try
    Modal.ShowModal;
  finally
    Modal.Free;
  end;
```

Because the `ShowModal` call can raise an exception, you should write it in a `try` block followed by a `finally` block to make sure the object will be de-allocated. Usually this block also includes code that initializes the dialog box before displaying it and code that extracts the values set by the user before destroying the form. The final values are read-only if the result of the `ShowModal` function is mrOK, as you'll see in the next example.

The situation is a little more complex when you want to display only one copy of a modeless form. You have to create the form, if it is not already available, and then show it:

```
if not Assigned (Form2) then
  Form2 := TForm2.Create (Application);
Form2.Show;
```

With this code, the form is created the first time it is required and then is kept in memory, visible on the screen or hidden from view. To avoid using up memory and system resources unnecessarily, you'll want to destroy the secondary form when it is closed. You can do that by writing a handler for the `OnClose` event:

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  // important: set pointer to nil!
  Form2 := nil;
end;
```

Notice that after you destroy the form, the global `Form2` variable is set to `nil`, which contradicts the rule set earlier for forms with multiple instances, but as this is a single-instance we are in the exact opposite case. Without this code, closing the form would destroy its object, but the `Form2` variable would still refer to the original memory location. At this point, if you try to show the form once more with the `btnSingleClick` method shown earlier, the `if not Assigned()` test will succeed, because it checks whether the `Form2` variable is `nil`. The code fails to create a new object, and the `Show` method (invoked on a nonexistent object) will result in a system memory error.

As an experiment, you can generate this error by removing the last line of the previous listing. As you have seen, the solution is to set the `Form2` object to `nil` when the object is destroyed, so that properly written code will "see" that a new form must be created before using it. Again, experimenting with the MultiWin/QMultiWin example can prove useful to test various conditions. (I haven't shown any screens from this example because the forms it displays are totally empty, except for the main form, which has three buttons.)

NOTE *Setting the form variable to* `nil` *makes sense—and works—if there is to be only one instance of the form present at any given instant. If you want to create multiple copies of a form, you'll have to use other techniques to keep track of them. Also keep in mind that in this case you cannot use the* `FreeAndNil` *procedure, because you cannot call* `Free` *on* `Form2`*—you cannot destroy the form before its event handlers have finished executing.*

Creating a Dialog Box

I stated earlier in this chapter that a dialog box is not very different from other forms. To build a dialog box instead of a form, you just select the bsDialog value for the `BorderStyle` property. With this simple change, the interface of the form becomes like that of a dialog box, with no system icon and no Minimize and Maximize boxes. Of course, such a form has the typical thick dialog box border, which is non-resizable.

Once you have built a dialog box form, you can display it as a modal or modeless window using the two usual show methods (`Show` and `ShowModal`). Modal dialog boxes, however, are more common than modeless ones. This is the reverse of forms; modal forms should generally be avoided, because a user won't expect them.

The Dialog Box of the RefList Example

In Chapter 5, "Visual Controls," we explored the RefList/QRefList program, which used a ListView control to display references to books, magazines, websites, and more. In the RefList2 version (and its QRefList2 CLX counterpart), I added to the basic version a dialog box that's used in two different circumstances: adding new items to the list and editing existing items.

WARNING *The CLX ListView component has a problem. In case you activate the check boxes and then disable them, the images will disappear. This is the behavior of the QRefList example of Chapter 5. In the QRefList2 version I've added code to reassign the* `ImageIndex` *property of each item as a workaround to this bug.*

The only particularly interesting feature of this form in the VCL example is the use of the ComboBoxEx component, which is attached to the same ImageList used by the ListView control of the main form. The drop-down items of the list, used to select a type of reference, include both a textual description and the corresponding image.

As I mentioned, this dialog box is used in two different cases. The first takes place as the user selects File ➢ Add Items from the menu:

```
procedure TForm1.AddItems1Click(Sender: TObject);
var
  NewItem: TListItem;
begin
  FormItem.Caption := 'New Item';
  FormItem.Clear;
  if FormItem.ShowModal = mrOK then
  begin
    NewItem := ListView1.Items.Add;
    NewItem.Caption := FormItem.EditReference.Text;
    NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
    NewItem.SubItems.Add (FormItem.EditAuthor.Text);
    NewItem.SubItems.Add (FormItem.EditCountry.Text);
  end;
end;
```

Besides setting the proper caption for the form, this procedure initializes the dialog box, because you are entering a new value. If the user clicks OK, however, the program adds a new item to the list view and sets all its values. To empty the dialog's edit boxes, the program calls the custom `Clear` method, which resets the text of each edit box control:

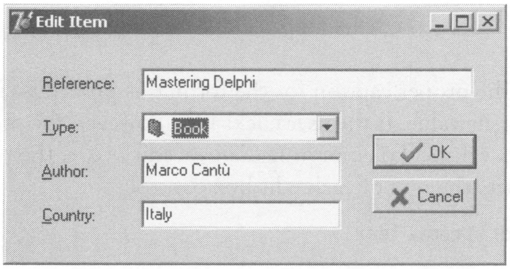
```
procedure TFormItem.Clear;
var
  I: Integer;
```

```
begin
  // clear each edit box
  for I := 0 to ControlCount - 1 do
    if Controls [I] is TEdit then
      TEdit (Controls[I]).Text := '';
end;
```

Editing an existing item requires a slightly different approach. First, the current values are moved to the dialog box before it is displayed. Second, if the user clicks OK, the program modifies the current list item instead of creating a new one. Here is the code:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
begin
  if ListView1.Selected <> nil then
  begin
    // dialog initialization
    FormItem.Caption := 'Edit Item';
    FormItem.EditReference.Text := ListView1.Selected.Caption;
    FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
    FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
    FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

    // show it
    if FormItem.ShowModal = mrOK then
    begin
      // read the new values
      ListView1.Selected.Caption := FormItem.EditReference.Text;
      ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
      ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
      ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
    end;
  end;
end;
```

You can see the effect of this code in Figure 7.11. Notice that the code used to read the value of a new item or modified item is similar. In general, you should try to avoid this type of duplicated code and perhaps place the shared code statements in a method added to the dialog box. In this case, the method could receive as parameter a `TListItem` object and copy the proper values into it.



Reference: Mastering Delphi

Type: Book

Author: Marco Cantù

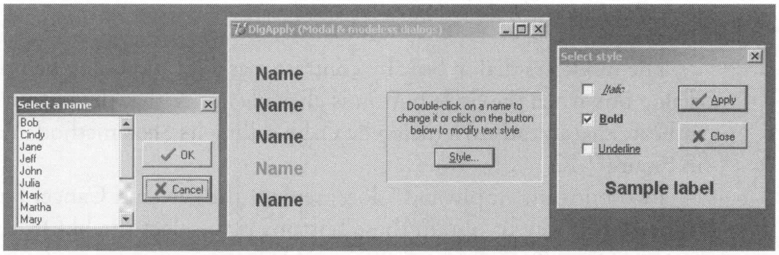
Country: Italy

FIGURE 7.11
The dialog box of the RefList2 example used in edit mode. Notice the ComboBoxEx graphical component in use.

NOTE *What happens internally when the user clicks the OK or Cancel button in the dialog box? A modal dialog box is closed by setting its* `ModalResult` *property, and it returns the value of this property. You can indicate the return value by setting the* `ModalResult` *property of the button. When the user clicks the button, its* `ModalResult` *value is copied to the form, which closes the form and returns the value as the result of the* `ShowModal` *function.*

A Modeless Dialog Box

The second example of dialog boxes shows a more complex modal dialog box that uses the standard approach as well as a modeless dialog box. The main form of the DlgApply example (and of the identical CLX-based QDlgApply demo) has five labels with names, as you can see in Figure 7.12 and by viewing the source code of the example.



Double-click on a name to change it or click on the button below to modify text style

FIGURE 7.12
The three forms (a main form and two dialog boxes) of the DlgApply example at run time

If the user clicks a name, its color changes to red; if the user double-clicks it, the program displays a modal dialog box with a list of names to choose from. If the user clicks the Style button, a modeless dialog box appears, allowing the user to change the font style of the main form's labels. The five labels on the main form are connected to two methods, one for the `OnClick` event and the second for the `OnDoubleClick` event. The first method turns the last label a user clicked red, resetting all the others to black (they have the `Tag` property set to 1, as a sort of group index). Notice that the same method is associated with all the labels:

```
procedure TForm1.LabelClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount - 1 do
   if (Components[I] is TLabel) and (Components[I].Tag = 1) then
     TLabel (Components[I]).Font.Color := clBlack;
  // set the color of the clicked label to red
  (Sender as TLabel).Font.Color := clRed;
end;
```

The second method common to all the labels is the `OnDoubleClick` event handler. The `LabelDoubleClick` method selects the `Caption` of the current label (indicated by the `Sender` parameter) in the list box of the dialog and then shows the modal dialog box. If the user closes the dialog box by clicking OK and a list item is selected, the selection is copied back to the label's caption:

```
procedure TForm1.LabelDoubleClick(Sender: TObject);
begin
  with ListDial.Listbox1 do
  begin
    // select the current name in the list box
    ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
    // show the modal dialog box, checking the return value
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
      // copy the selected item to the label
      (Sender as TLabel).Caption := Items [ItemIndex];
  end;
end;
```

TIP *Notice that all the code used to customize the modal dialog box is in the* `LabelDoubleClick` *method of the main form. The form of this dialog box has no added code.*

The modeless dialog box, by contrast, has a lot of coding behind it. The main form displays the dialog box when the Style button is clicked (notice that the button caption ends with three dots to indicate that it leads to a dialog box), by calling its `Show` method. You can see the dialog box running in Figure 7.12.

Two buttons, Apply and Close, replace the OK and Cancel buttons in a modeless dialog box. (The fastest way to obtain these buttons is to select the bkOK or bkCancel value for the `Kind` property and then edit the `Caption`.) At times, you may see a Cancel button that works as a Close button, but the OK button in a modeless dialog box usually has no meaning. Instead, one or more buttons might perform specific actions on the main window, such as Apply, Change Style, Replace, Delete, and so on.

If the user clicks one of the check boxes in this modeless dialog box, the style of the sample label's text at the bottom changes accordingly. You accomplish this by adding or removing the specific flag that indicates the style, as in the following `OnClick` event handler:

```
procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);
begin
  if ItalicCheckBox.Checked then
    LabelSample.Font.Style := LabelSample.Font.Style + [fsItalic]
  else
    LabelSample.Font.Style := LabelSample.Font.Style - [fsItalic];
end;
```

When the user clicks the Apply button, the program copies the style of the sample label to each of the form's labels, rather than consider the values of the check boxes:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
begin
  Form1.Label1.Font.Style := LabelSample.Font.Style;
  Form1.Label2.Font.Style := LabelSample.Font.Style;
  ...
```

As an alternative, instead of referring to each label directly, you can look for it by calling the `FindComponent` method of the form, passing the label name as a parameter, and then casting the result to the `TLabel` type. The advantage of this approach is that you can create the names of the various labels with a `for` loop:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    (Form1.FindComponent ('Label' + IntToStr (I)) as TLabel).Font.Style :=
      LabelSample.Font.Style;
end;
```

TIP *The* `ApplyBitBtnClick` *method could also be written by scanning the* `Controls` *array in a loop, as I've done in other examples. I decided to use the* `FindComponent` *method here to demonstrate a different technique.*

This second version of the code is certainly slower, because it has more operations to do, but you won't notice the difference because it is still very fast. Of course, this approach is also more flexible; if you add a new label, you only need to fix the higher limit of the `for` loop, provided all the labels have consecutive numbers.

Notice that when the user clicks the Apply button, the dialog box does not close—only the Close button has this effect. Consider also that this dialog box needs no initialization code because the form is not destroyed, and its components maintain their status each time the dialog box is displayed. Notice, however, that in the CLX version of the program, QDlgApply, the dialog is modal, even if it is called with the `Show` method.

Predefined Dialog Boxes

Besides building your own dialog boxes, Delphi allows you to use some default dialog boxes of various kinds. Some are predefined by Windows; others are simple dialog boxes (such as message boxes) displayed by a Delphi routine. The Delphi Component Palette contains a page of dialog box components. Each of these dialog boxes—known as *Windows common dialogs*—is defined in the system library `ComDlg32.DLL`.

Windows Common Dialogs

I have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the `Execute` method, returning a Boolean value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, I've built the CommDlgTest program.

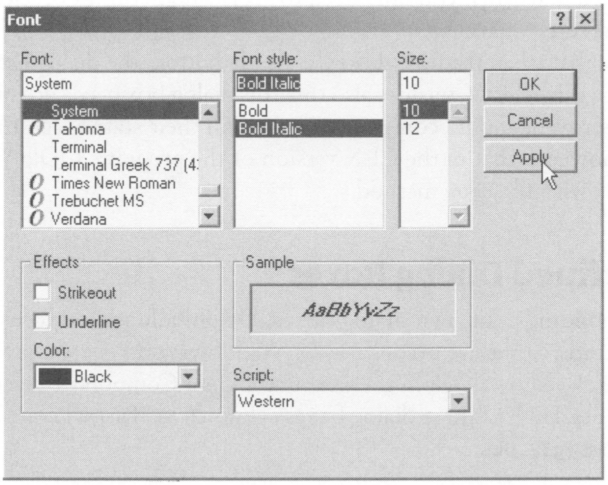
I'll highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The Open Dialog Component can be customized by setting different file extension filters using the `Filter` property, which has a handy editor and can be assigned a value directly with a string like `Text File (*.txt)|*.txt`. Another useful feature lets the dialog check whether the extension of the selected file matches the default extension, by checking the `ofExtensionDifferent` flag of the `Options` property after executing the dialog. Finally, this dialog allows multiple selections by setting its ofAllowMultiSelect option. In this case you can get the list of selected files by looking at the `Files` string list property.

- The SaveDialog component is used in similar ways and has similar properties, although of course you cannot select multiple files.
- The OpenPictureDialog and SavePictureDialog components provide similar features but have a customized form that shows a preview of an image. It only makes sense to use these components for opening or saving graphical files.
- The FontDialog component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (WYSIWYG), or only TrueType fonts. You can show or hide the portion related to the special effects, and obtain other different versions by setting its `Options` property. You can also activate an Apply button by providing an event handler for its `OnApply` event and using the fdApplyButton option. A Font dialog box with an Apply button (see Figure 7.13) behaves almost like a modeless dialog box (but isn't one).

FIGURE 7.13

The Font selection dialog box with an Apply button



Font: System

Font style: Bold Italic

Size: 10

☐ Strikeout

☐ Underline

Color: Black

Script: Western

- The ColorDialog component is used with different options to show the dialog fully open at first or to prevent it from opening fully. These settings are the cdFullOpen or cdPreventFullOpen values of the `Options` property.
- The Find and Replace dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the CommDlgTest example. The custom code is connected to the buttons of the two dialog boxes by providing the `OnFind` and `OnReplace` events.

NOTE *Qt offers a similar set of predefined dialog boxes, but the set of options is often more limited. I've created the QCommDlg version of the example you can use to experiment with these settings. The CLX program has fewer menu items, because some of the options are not available; there are other minimal changes in the source code.*

A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. You can use many Delphi procedures and functions to display simple dialog boxes:

- The `MessageDlg` function shows a customizable message box with one or more buttons and usually a bitmap. The `MessageDlgPos` function is similar to the `MessageDlg` function, but the message box is displayed in a given position, not in the center of the screen (unless you use the –1, –1 position to make it appear in the screen center).
- The `ShowMessage` procedure displays a simpler message box with the application name as the caption and an OK button. The `ShowMessagePos` procedure does the same, but you also indicate the position of the message box. The `ShowMessageFmt` procedure is a variation of `ShowMessage`, which has the same parameters as the `Format` function. It corresponds to calling `Format` inside a call to `ShowMessage`.
- The `MessageBox` method of the `Application` object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the `MessageBox` function of the Windows API, which passes as a main window parameter the handle of the `Application` object. This handle is required to make the message box behave like a modal window.
- The `InputBox` function asks the user to input a string. You provide the caption, the query, and a default string. The `InputQuery` function asks the user to input a string, too. The only difference between this and the `InputBox` function is in the syntax. The `InputQuery` function has a Boolean return value that indicates whether the user has clicked OK or Cancel.

To demonstrate some of the message boxes available in Delphi, I've written another sample program, with a similar approach to the preceding CommDlgTest example. In the MBParade example, you have a high number of choices (radio buttons, check boxes, edit boxes, and spin edit controls) to set before you click one of the buttons that displays a message box. The similar QMbParade example is missing only the Help button, which is not available in the CLX message boxes.

About Boxes and Splash Screens

Applications usually have an About box in which you can display information such as the version of the product, a copyright notice, and so on. The simplest way to build an About box is to use the `MessageDlg` function. With this method, you can show only a limited amount of text and no special graphics.

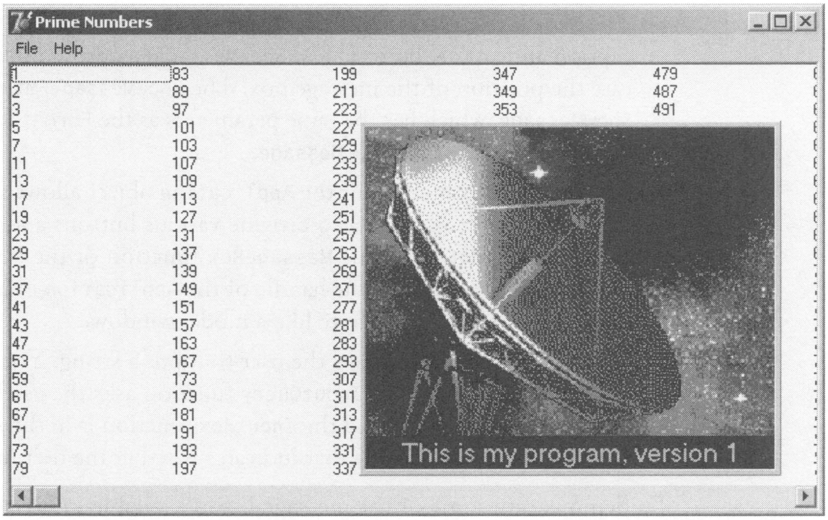
Therefore, the usual method for creating an About box is to use a dialog box, such as the one generated with one of the Delphi default templates. In this About box, you might want to add some code to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

Building a Splash Screen

Another typical technique displays an initial screen before the application's main form is shown. Doing so makes the application seem more responsive, because you show something to the user while the program is loading, and it also makes a nice visual effect. Sometimes this same window is displayed as the application's About box. For an example in which a splash screen is particularly useful, I've built a program displaying a list box filled with prime numbers.

The prime numbers are computed on program startup, with a `for` loop running from 1 to 30,000; the numbers are displayed as soon as the form becomes visible. Because I've used (on purpose) a slow function to compute prime numbers, this initialization code takes quite some time. The numbers are added to a list box that covers the full client area of the form and allows multiple columns to be displayed, as you can see in Figure 7.14.

FIGURE 7.14
The main form of the Splash example, with the splash screen (this is the Splash2 version)



```
1    83   199  347  479
2    89   211  349  487
3    97   223  353  491
5    101  227
7    103  229
11   107  233
13   109  239
17   113  241
19   127  251
23   131  257
29   137  263
31   139  269
37   149  271
41   151  277
43   157  281
47   163  283
53   167  293
59   173  307
61   179  311
67   181  313
71   191  317
73   193  331
79   197  337
```

This is my program, version 1

There are three versions of the Splash program (plus the three corresponding CLX versions). As you can see by running the Splash0 example, the problem with this program is that the initial operation, which takes place in the `FormCreate` method, takes a lot of time. When you start the program, it takes several seconds to display the main form. If your computer is very fast or very slow, you can change the upper limit of the `for` loop in the `FormCreate` method to make the program faster or slower.

This program has a simple dialog box with an image component, a caption, and a bitmap button, all placed inside a panel taking up the whole surface of the About box. This form is displayed when you select the Help ➢ About menu item. But you really want to display this About box while the program starts. You can see this effect by running the Splash1 and Splash2 examples, which show a splash screen using two different techniques.

First, I've added a method to the `TAboutBox` class. This method, called `MakeSplash`, changes some properties of the form to make it suitable for a splash form. Basically, it removes the border and caption, hides the OK button, makes the border of the panel thick (to replace the border of the form), and then shows the form, repainting it immediately:

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
```

```
    Panel1.BorderWidth := 3;
    Show;
    Update;
  end;
```

This method is called after creating the form in the project file of the Splash1 example. This code is executed before creating the other forms (in this case only the main form), and the splash screen is then removed before running the application. These operations take place within a `try/finally` block. Here is the source code of the main block of the project file for the Splash2 example:

```
var
  SplashAbout: TAboutBox;

begin
  Application.Initialize;

  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  try
    SplashAbout.MakeSplash;
    // standard code...
    Application.CreateForm(TForm1, Form1);
    // get rid of the splash form
    SplashAbout.Close;
  finally
    SplashAbout.Free;
  end;

  Application.Run;
end.
```

This approach makes sense only if your application's main form takes a while to create, to execute its startup code (as in this case), or to open database tables. Notice that the splash screen is the first form created, but because the program doesn't use the `Application` object's `CreateForm` method, it doesn't become the main form of the application. In this case, closing the splash screen would terminate the program!

An alternative approach is to keep the splash form on the screen a little longer and use a timer to get rid of it. I've implemented this technique in the Splash2 example. This example also uses a different approach for creating the splash form: Instead of creating the splash form in the project source code, it creates the form at the very beginning of the `FormCreate` method of the main form.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  SplashAbout: TAboutBox;
begin
  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  SplashAbout.MakeSplash;
```

```
    // slow code (omitted)...
    // get rid of the splash form, after a while
    SplashAbout.Timer1.Enabled := True;
  end;
```

The timer is enabled just before terminating the method. After its interval has elapsed (in the example, 3 seconds) the `OnTimer` event is activated, and the splash form handles it by closing and destroying itself, calling `Close` and then `Release`.

NOTE *The `Release` method of a form is similar to the `Free` method of objects, but the destruction of the form is delayed until all event handlers have completed execution. Using `Free` inside a form might cause an access violation, because the internal code that fired the event handler might refer again to the form object.*

There is one more thing to fix. The main form will be displayed later and in front of the splash form, unless you make it a top-most form. For this reason, I've added one line to the `MakeSplash` method of the About box in the Splash2 example:

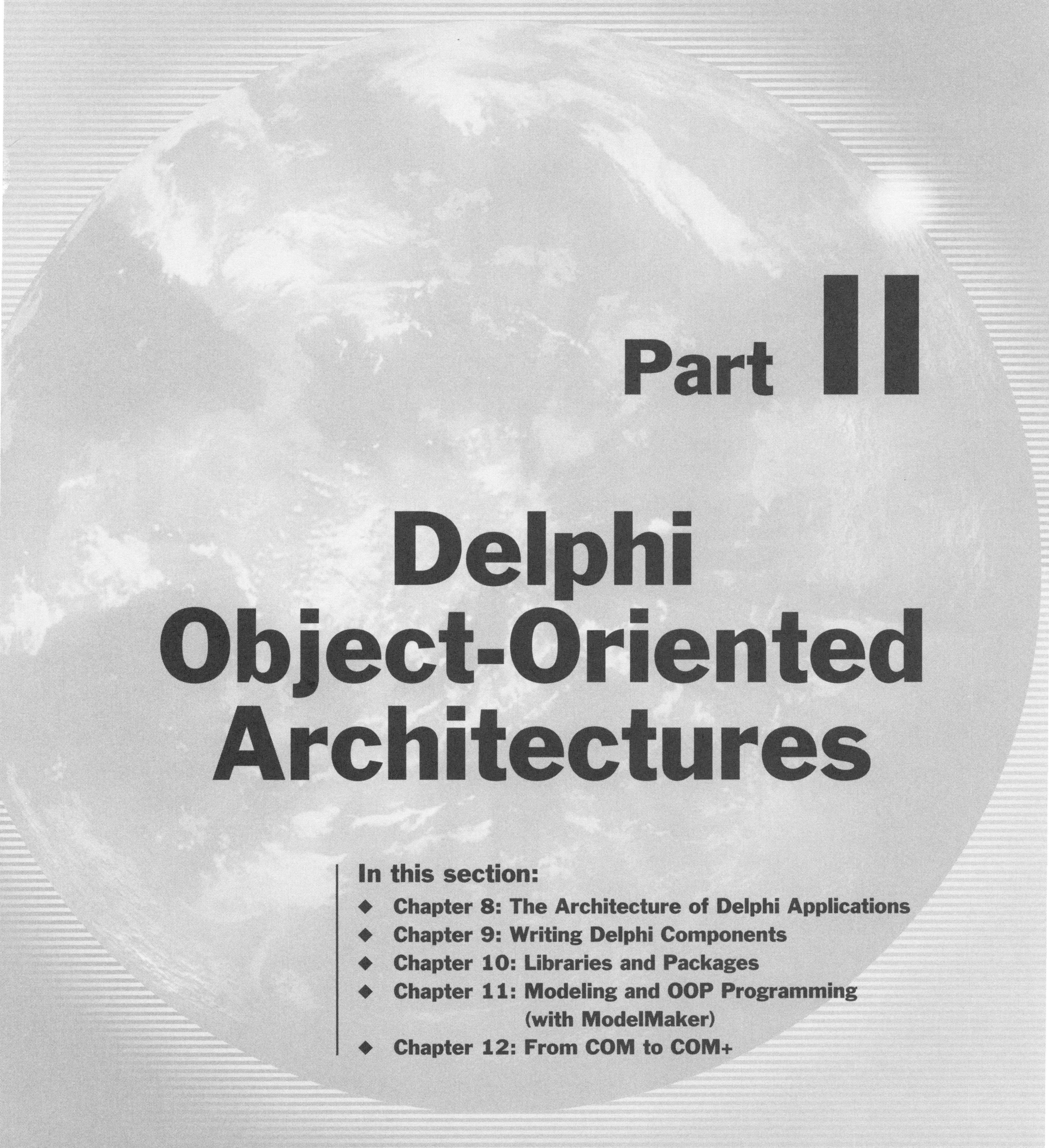
```
  FormStyle := fsStayOnTop;
```

What's Next?

In this chapter, we've explored some important form properties. Now you know how to handle the size and position of a form, how to resize it, and how to get mouse input and paint over it. You know more about dialog boxes, modal forms, predefined dialogs, splash screens, and many other techniques, including the funny effect of alpha blending. Understanding the details of working with forms is critical to proper use of Delphi, particularly for building complex applications (unless, of course, you're building services or web applications with no user interface).

In Chapter 8, we'll continue by exploring the overall structure of a Delphi application, with coverage of the role of two global objects: `Application` and `Screen`. I'll also discuss MDI development as you learn about more advanced features of forms, such as visual form inheritance. In addition, I'll discuss frames, which are visual component containers similar to forms.

In this chapter, I've also provided a short introduction to direct painting and to the use of the `TCanvas` class. More about graphics in Delphi forms can also be found in the bonus chapter "Graphics in Delphi", discussed in Appendix C.



Part II

Delphi Object-Oriented Architectures

In this section:

- Chapter 8: The Architecture of Delphi Applications
- Chapter 9: Writing Delphi Components
- Chapter 10: Libraries and Packages
- Chapter 11: Modeling and OOP Programming (with ModelMaker)
- Chapter 12: From COM to COM+

Chapter 8

The Architecture of Delphi Applications

ALTHOUGH TOGETHER WE'VE BUILT Delphi applications since the beginning of the book, we haven't focused on the structure and the architecture of an application built with Delphi's class libraries. For example, I haven't included much coverage about the global `Application` object, techniques for keeping tracks of forms you've created, the flow of messages in the system, and other such elements.

In Chapter 7, "Working with Forms," you saw how to create applications with multiple forms and dialog boxes. However, I haven't discussed how these forms can be related one to the other, how you can share similar features of forms, and how you can operate on multiple similar forms in a consistent way.

My ambitious goal in this chapter is to discuss all these topics. The chapter covers both basic and advanced techniques, including visual form inheritance, the use of frames, and MDI development, as well as the use of interfaces for building complex hierarchies of form classes.

This chapter covers the following:

- The `Application` and `Screen` global objects
- Messages and multitasking in Windows
- Background processing and multithreading
- Finding the previous instance of an application
- MDI applications
- Visual form inheritance
- Frames
- Base forms and interfaces
- Delphi's memory manager

The *Application* Object

I've mentioned the `Application` global object on multiple occasions, but because this chapter focuses on the structure of Delphi applications, it is time to delve into the details of this global object and its corresponding class. `Application` is a global object of the `TApplication` class, defined in the Forms unit and created in the Controls unit. The `TApplication` class is a component, but you cannot use it at design time. Some of its properties can be directly set in the Application page of the Project Options dialog box; others must be assigned in code.

To handle its events, Delphi includes a handy ApplicationEvents component. Besides allowing you to assign handlers at design time, the advantage of this component is that it allows for multiple handlers. If you simply place an instance of the ApplicationEvents component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple ApplicationEvents components can chain the handlers.

Some of these application-wide events, including `OnActivate`, `OnDeactivate`, `OnMinimize`, and `OnRestore`, allow you to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as in `OnActionExecute`, `OnActionUpdate`, `OnHelp`, `OnHint`, `OnShortCut`, and `OnShowHint`. Finally, there is the `OnException` global exception handler we used in Chapter 2 ("The Delphi Programming Language"), the `OnIdle` event used for background computing, and the `OnMessage` event, which fires when a message is posted to any of the windows or windowed controls of the application.

Although its class inherits directly from `TComponent`, the `Application` object has a window associated with it. The application window is hidden from sight but appears on the Taskbar. This is why Delphi names the window *Form1* and the corresponding Taskbar icon *Project1*.

The window related to the `Application` object—the application window—serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this invisible owner window, for example, is fundamental when the application is activated. When your program's windows are behind other programs' windows, clicking one window in your application will bring all of that application's windows to the front. In other words, the unseen application window is used to connect the application's various forms. (The application window is not *hidden*, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.)

TIP *In Windows, the Minimize and Maximize operations are associated by default with system sounds and a visual animated effect. Applications built with Delphi produce the sound and display the visual effect by default.*

When you create a new, blank application, Delphi generates code for the project file that includes the following:

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

As you can see in this standard code, the `Application` object can create forms, setting the first one as the `MainForm` (one of the `Application` properties) and closing the entire application when this main form is destroyed. The program execution is enclosed in the `Run` method, which embeds the system loop to process system messages. This loop continues until the application's main window (the first window you created) is closed.

TIP *As we saw in the splash screen example in Chapter 7, the main form is not necessarily the first form you create, but the first one you create by calling* `Application.CreateForm`.

The Windows message loop embedded in the `Run` method delivers the system messages to the proper application windows. A message loop is required by any Windows application, but you don't need to write one in Delphi because the `Application` object provides a default loop.

In addition to performing this main role, the `Application` object manages a few other interesting areas:

- Hints (discussed at the end of Chapter 5, "Visual Controls")
- The help system, which includes the ability to define the type of help viewer (a topic not covered in detail in this book)
- Application activation, minimization, and restoration
- A global exceptions handler, as discussed in the ErrorLog example of Chapter 2.
- General application information, including the `MainForm`, executable filename and path (`ExeName`), `Icon`, and `Title` displayed in the Windows Taskbar and when you scan the running applications with the Alt+Tab keys

TIP *To avoid a discrepancy between the two titles, you can change the application's title at design time. In case the caption of the main form changes at runtime, you can copy it to the title of the application with this code:* `Application.Title := Form1.Caption`.

In most applications, you don't care about the application window, apart from setting its `Title` and icon and handling some of its events. However, you can perform some other simple operations. Setting the `ShowMainForm` property to False in the project source code indicates that the main form should not be displayed at startup. Inside a program, you can use the `Application` object's `MainForm` property to access the main form.

Displaying the Application Window

There is no better proof that a window exists for the `Application` object than to display it, an in the ShowApp example. You don't need to show it—you just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. You can perform these operations using Windows API functions on the window indicated by the `Application` object's `Handle` property:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldStyle: Integer;
begin
  // add border and caption to the app window
  OldStyle := GetWindowLong (Application.Handle, gwl_Style);
  SetWindowLong (Application.Handle, gwl_Style,
    OldStyle or ws_ThickFrame or ws_Caption);
  // set the size of the app window
  SetWindowPos (Application.Handle, 0, 0, 0, 200, 100,
    swp_NoMove or swp_NoZOrder);
end;
```

The `GetWindowLong` and `SetWindowLong` API functions access the system information related to the window. In this case, you use the `gwl_Style` parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. This code gets the current styles and adds (using an `or` statement) a standard border and a caption to the form.

Of course, you generally won't need to implement something like this in your programs. But knowing the application object has a window connected to it is an important aspect of understanding the default structure of Delphi applications and being able to modify it when needed.

Activating Applications and Forms

To show how the activation of forms and applications works, I've written a self-explanatory example called ActivApp. This example has two forms. Each form has a Label component (`LabelForm`) used to display the form's status. The program uses text and color to indicate this status information, as the handlers of the first form's `OnActivate` and `OnDeactivate` events demonstrate:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Active';
  LabelForm.Color := clRed;
end;

procedure TForm1.FormDeactivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Not Active';
  LabelForm.Color := clBtnFace;
end;
```

The second form has a similar label and similar code.

The main form also displays the status of the entire application. It uses an ApplicationEvents component to handle the `Application` object's `OnActivate` and `OnDeactivate` events. These two event handlers are similar to the two listed previously; the only difference is that they modify the text and color of a second label on the form and that one of them makes a beep.

If you run this program, you'll see whether this application is active and, if so, which of its forms is active. By looking at the output (see Figure 8.1) and listening for the beep, you can understand how Delphi triggers each of the activation events. Run the program and play with it for a while to understand how it works. Later, we'll get back to other events related to the activation of forms.

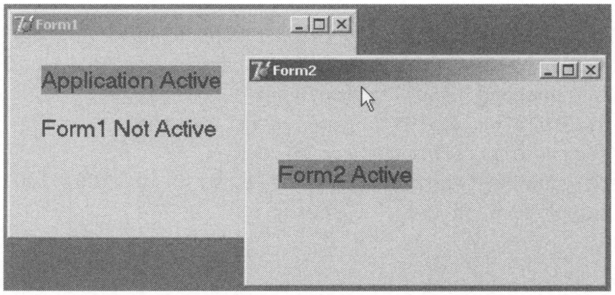


FIGURE 8.1
The ActivApp example shows whether the application is active and which of the application's forms is active.

Tracking Forms with the *Screen* Object

We have already explored some of the properties and events of the `Application` object. Other interesting global information about an application is available through the `Screen` object, whose base class is `TScreen`. This object holds information about the system display (the screen size and the screen fonts) and also about the current set of forms in a running application. For example, you can display the screen size and the list of fonts by writing:

```
Label1.Caption := IntToStr (Screen.Width) + 'x' + IntToStr (Screen.Height);
ListBox1.Items := Screen.Fonts;
```

`TScreen` also reports the number and resolution of monitors in a multimonitor system. Right now, however, I will focus on the list of forms held by the `Screen` object's `Forms` property, the top-most form indicated by the `ActiveForm` property, and the related `OnActiveFormChange` event. Note that the forms the `Screen` object references are the forms of the application and not those of the system.

These features are demonstrated by the Screen example, which maintains a list of the current forms in a list box. This list must be updated each time a new form is created, an existing form is destroyed, or the program's active form changes. To see how this process works, you can create secondary forms by clicking the button labeled New:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
var
  NewForm: TSecondForm;
begin
  // create a new form, set its caption, and run it
  NewForm := TSecondForm.Create (Self);
  Inc (nForms);
  NewForm.Caption := 'Second ' + IntToStr (nForms);
  NewForm.Show;
end;
```

Note that you need to disable the automatic creation of the secondary form by using the Forms page of the Project Options dialog box. One of the key portions of the program is the form's `OnCreate` event handler, which fills the list the first time and then connects a handler to the `OnActiveFormChange` event:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FillFormsList (Self);
  // set the secondary form's counter to 0
  nForms := 0;
  // set an event handler on the screen object
  Screen.OnActiveFormChange := FillFormsList;
end;
```

The code used to fill the Forms list box is inside a second procedure, `FillFormsList`, which is also installed as an event handler for the `Screen` object's `OnActiveFormChange` event:

```
procedure TMainForm.FillFormsList (Sender: TObject);
var
  I: Integer;
```

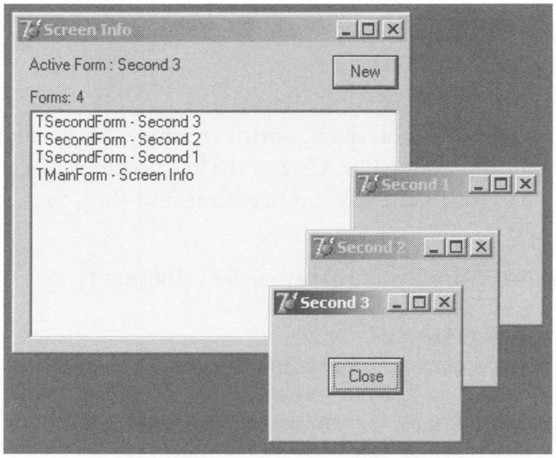
```
begin
  // skip code in destruction phase
  if Assigned (FormsListBox) then
  begin
    FormsLabel.Caption := 'Forms: ' + IntToStr (Screen.FormCount);
    FormsListBox.Clear;
    // write class name and form title to the list box
    for I := 0 to Screen.FormCount - 1 do
      FormsListBox.Items.Add (Screen.Forms[I].ClassName + ' - ' +
        Screen.Forms[I].Caption);
    ActiveLabel.Caption := 'Active Form : ' + Screen.ActiveForm.Caption;
  end;
end;
```

WARNING *It is very important not to execute this code while the main form is being destroyed. As an alternative to testing whether the list box is set to* `nil`*, you could test the form's* `ComponentState` *for the csDestroying flag. Another approach would be to remove the* `OnActiveFormChange` *event handler before exiting the application; that is, handle the main form's* `OnClose` *event and assign* `nil` *to* `Screen.OnActiveFormChange`*.*

The `FillFormsList` method fills the list box and sets a value for the two labels above it to show the number of forms and the name of the active form. When you click the New button, the program creates an instance of the secondary form, gives it a new title, and displays it. The Forms list box is updated automatically because of the handler installed for the `OnActiveFormChange` event. Figure 8.2 shows the output of this program when some secondary windows have been created.

FIGURE 8.2

The output of the Screen example with some secondary forms



```
Active Form : Second 3
Forms: 4
TSecondForm - Second 3
TSecondForm - Second 2
TSecondForm - Second 1
TMainForm - Screen Info
```

Each secondary form has a Close button you can click to remove it. The program handles the `OnClose` event, setting the `Action` parameter to caFree, so that the form is destroyed when it is closed. This code closes the form, but it doesn't update the list of the windows properly. The system moves the focus to another window first, firing the event that updates the list, and destroys the old form only after this operation.

The first idea I had to update the windows list properly was to introduce a delay, posting a user-defined Windows message. Because the posted message is queued and not handled immediately, if you send it at the last possible moment of the secondary form's life, the main form will receive it when the other form is destroyed. The trick is to post the message in the secondary form's `OnDestroy` event handler. To accomplish this, you need to refer to the `MainForm` object by adding a `uses` statement in the implementation portion of this unit. I've posted a `wm_User` message, which is handled by a specific `message` method of the main form, as shown here:

```
public
  procedure ChildClosed (var Message: TMessage); message wm_User;

procedure TMainForm.ChildClosed (var Message: TMessage);
begin
  FillFormsList (Self);
end;
```

The problem is that if you close the main window before closing the secondary forms, the main form exists, but its code can no longer be executed. To avoid another system error (an Access Violation Fault), you need to post the message only if the main form is not closing. But how do you determine whether the form is closing? One way is to add a flag to the `TMainForm` class and change its value when the main form is closing, so that you can test the flag from the code of the secondary window.

This is a good solution—so good that the VCL already provides similar functionality with the `ComponentState` property and its csDestroying flag, as mentioned earlier. Therefore, you can write the following code:

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

With this code, the list box always lists all the forms in the application.

After giving this approach some thought, however, I found an alternative and much more Delphi-oriented solution. The trick is to consider that every time a component is destroyed, it tells its owner about the event by calling the `Notification` method defined in the `TComponent` class. Because the secondary forms are owned by the main form, as specified in the `NewButtonClick` method's code, you can override this method and simplify the code (see the Screen2 folder for this version's code):

```
procedure TMainForm.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and Showing and (AComponent is TForm) then
    FillFormsList;
end;
```

NOTE *If the secondary forms were not owned by the main form, you could have used the* `FreeNotification` *method to get the secondary forms to notify the main form when they are destroyed.* `FreeNotification` *receives as parameter the component to notify when the current component is destroyed. The effect is a call to the* `Notification` *method that comes from a component other than the owned components.* `FreeNotification` *is generally used by component writers to safely connect components on different forms or data modules.*

The last feature I've added to both versions of the program is simple: When you click an item in the list box, the corresponding form is activated using the `BringToFront` method. Nice—well, almost nice. If you click the list box when the main form is not active, the main form is activated first, and the list box is rearranged; so, you might end up selecting a different form than you were expecting. If you experiment with the program, you'll soon realize what I mean. This minor glitch in the program is an example of the risks you face when you dynamically update information and let the user work on it at the same time.

From Events to Threads

To understand how Windows applications work internally, let's spend a minute discussing how multitasking is supported in this environment. You also need to understand the role of timers (and the Timer component) and of background (or *idle*) computing, as well as the `ProcessMessages` method of the `Application` global object.

In short, we need to delve deeper into the event-driven structure of Windows and its multitasking support. Because this is a book about *Delphi* programming, I won't discuss this topic in detail, but I will provide an overview for readers who have limited experience with Windows API programming.

Event-Driven Programming

The basic idea behind event-driven programming is that specific events determine the control flow of the application. A program spends most of its time waiting for these events and provides code to respond to them. For example, when a user clicks one of the mouse buttons, an event occurs. A message describing this event is sent to the window currently under the mouse cursor. The program code that responds to events for that window receives the event, processes it, and responds accordingly. When the program has finished responding to the event, it returns to a waiting or "idle" state.

As this explanation shows, events are serialized; each event is handled only after the previous one is completed. When an application is executing event-handling code (that is, when it is not waiting for an event), other events for that application have to wait in a message queue reserved for that application (unless the application uses multiple threads). When an application has responded to a message and returned to a waiting state, it becomes the last in the list of programs waiting to handle additional messages. In every version of Win32 (9*x*, NT, Me, and 2000), after a fixed amount of time has elapsed, the system interrupts the current application and immediately gives control to the next program in the list. The first program is resumed only after each application has had a turn. This process is called *preemptive multitasking*.

So, an application performing a time-consuming operation in an event handler doesn't prevent the system from working properly (because other processes have their time-slice of the CPU), but the application generally is unable even to repaint its own windows properly—with a very nasty effect. If you've never experienced this problem, try it for yourself: Write a time-consuming loop that executes when a button is clicked, and try to move the form or move another window on top of it. The effect is really annoying. Now try adding the call `Application.ProcessMessages` within the loop; you'll see that the operation becomes much slower, but the form will be refreshed immediately.

As an example of the use of `Application.ProcessMessages` within a time-consuming loop (and the lack of this call), you can refer to the BackTask example. Here is the code using this approach (ignore the naïve technique for computing the sum of a given set of prime numbers):

```
procedure TForm1.Button2Click(Sender: TObject);
var
  I, Tot: Integer;
begin
  Tot := 0;
  for I := 1 to Max do
  begin
    if IsPrime (I) then
      Tot := Tot + I;
    ProgressBar1.Position := I * 100 div Max;
    Application.ProcessMessages;
  end;
  ShowMessage (IntToStr (Tot));
end;
```

TIP There is a second alternative to calling `ProcessMessages`*: the* `HandleMessage` *function. There are two differences:* `HandleMessage` *processes at most one message each time it is called, whereas* `ProcessMessages` *keeps processing messages in the queue; and* `HandleMessage` *also activates idle time processing, such as action update calls.*

If an application has responded to its events and is waiting for its turn to process messages, it has no chance to regain control until it receives another message (unless it uses multithreading). This is a reason to use a *timer*: a system component that will send a message to your application whenever a specified time interval elapses. Using a timer is the only way to make an application perform operations automatically from time to time, even when the user is absent or not using the program (so that it is not processing any events).

One final note—when you think about events, remember that input events (generated using the mouse or the keyboard) account for only a small percentage of the total message flow in a Windows application. Most of the messages are the system's internal messages or messages exchanged between different controls and windows. Even a familiar input operation such as clicking a mouse button can result in a huge number of messages, most of which are internal Windows messages. You can test this yourself by using the WinSight utility included in Delphi. In WinSight, choose to view the Message Trace, and select the messages for all the windows. Click Start, and then perform some normal operations with the mouse. You'll see hundreds of messages in a few seconds.

Windows Message Delivery

Before looking at some real examples, let's consider another key element of message handling. Windows has two different ways to send a message to a window:

***PostMessage* API Function** Places a message in the application's message queue. The message will be handled only when the application has a chance to access its message queue (that is, when it receives control from the system), and only after earlier messages have been processed. This is an asynchronous call, because you do not know when the message will be received.

***SendMessage* API function** Executes message-handler code immediately. `SendMessage` bypasses the application's message queue and sends the message directly to a target window or control. This is a synchronous call. This function even has a return value, which is passed back by the message-handling code. Calling `SendMessage` is no different than directly calling another method or function of the program.

The difference between these two ways of sending messages is similar to that between mailing a letter, which will reach its destination sooner or later, and sending a fax, which goes immediately to the recipient. Although you will rarely need to use these low-level functions in Delphi, this description should help you determine which one to use if you do need to write this type of code.

Background Processing and Multitasking

Suppose you need to implement a time-consuming algorithm. If you write the algorithm as a response to an event, your application will be stopped completely during the time it takes to process that algorithm. To let the user know that something is being processed, you can display the hourglass cursor or show a progress bar, but this is not a user-friendly solution. Win32 allows other programs to continue their execution, but the program in question will appear to be frozen; it won't even update its own user interface if a repaint is requested. While the algorithm is executing, the application won't be able to receive and process any other messages, including paint messages.

The simplest solution to this problem is to call the `ProcessMessages` and `HandleMessage` methods, discussed earlier. The problem with this approach, however, is that the user might click the button again or re-press the keystrokes that started the algorithm. To fix this possibility, you can disable the buttons and commands you don't want the user to select, and you can display the hourglass cursor (which technically doesn't prevent a mouse-click event, but does suggest that the user should wait before doing any other operation).

For some low-priority background processing, you can also split the algorithm into smaller pieces and execute each of them in turn, letting the application fully respond to pending messages in between processing the pieces. You can use a timer to let the system notify you once a time interval has elapsed. Although you can use timers to implement some form of background computing, this is far from a good solution. A better technique would be to execute each step of the program when the `Application` object receives the `OnIdle` event.

The difference between calling `ProcessMessages` and using the `OnIdle` event is that calling `ProcessMessages` gives your code more processing time. Calling `ProcessMessages` lets the program perform other operations while a long operation is being executed; using the `OnIdle` event lets your application perform background tasks when it doesn't have pending requests from the user.

Delphi Multithreading

When you need to perform background operations or any processing not strictly related to the user interface, you can follow the technically most correct approach: spawn a separate thread of execution within the process. Multithreading programming might seem like an obscure topic, but it really isn't that complex, even if you must consider it with care. It is worth knowing at least the basics of multithreading, because in the world of sockets and Internet programming, there is little you can do without threads.

Delphi's RTL library provides a `TThread` class that will let you create and control threads. You will never use the `TThread` class directly, because it is an abstract class—a class with a virtual abstract method. To use threads, you always subclass `TThread` and use the features of this base class.

The `TThread` class has a constructor with a single parameter (`CreateSuspended`) that lets you choose whether to start the thread immediately or suspend it until later. If the thread object starts automatically, or when it is resumed, it will run its `Execute` method until it is done. The class provides a protected interface, which includes the two key methods for your thread subclasses:

```
procedure Execute; virtual; abstract;
procedure Synchronize(Method: TThreadMethod);
```

The `Execute` method, declared as a virtual abstract procedure, must be redefined by each thread class. It contains the thread's main code—the code you would typically place in a *thread function* when using the system functions.

The `Synchronize` method is used to avoid concurrent access to VCL components. The VCL code runs inside the program's main thread, and you need to synchronize access to VCL to avoid re-entry problems (errors from re-entering a function before a previous call is completed) and concurrent access to shared resources. The only parameter of `Synchronize` is a method that accepts no parameters, typically a method of the same thread class. Because you cannot pass parameters to this method, it is common to save some values within the data of the thread object in the `Execute` method and use those values in the *synchronized* methods.

NOTE *Delphi 7 includes two new versions of* `Synchronize` *that allow you to synchronize a method with the main thread without calling it from the thread object. Both the new overloaded* `Synchronize` *and* `StaticSynchronize` *are class methods of* `TThread` *and require a thread as parameter.*

Another way to avoid conflicts is to use the synchronization techniques offered by the operating system. The SyncObjs unit defines a few VCL classes for some of these low-level synchronization objects, such as events (with the `TEvent` class and the `TSingleEvent` class) and critical sections (with the `TCriticalSection` class). (Synchronization events should not be confused with Delphi events, as the two concepts are unrelated.)

AN EXAMPLE OF THREADING

For an example of a thread, you can refer again to the BackTask example. This example spawns a secondary thread for computing the sum of the prime numbers. The thread class has the typical `Execute` method, an initial value passed in a public property (`Max`), and two internal values (`FTotal` and `FPosition`) used to synchronize the output in the `ShowTotal` and `UpdateProgress` methods. The following is the complete class declaration for the custom thread object:

```
type
  TPrimeAdder = class(TThread)
  private
    FMax, FTotal, FPosition: Integer;
  protected
    procedure Execute; override;
    procedure ShowTotal;
    procedure UpdateProgress;
  public
    property Max: Integer read FMax write FMax;
  end;
```

The `Execute` method is very similar to the code used for the buttons in the BackTask example listed earlier. The only difference is in the final call to `Synchronize`, as you can see in the following two fragments:

```
procedure TPrimeAdder.Execute;
var
  I, Tot: Integer;
begin
  Tot := 0;
  for I := 1 to FMax do
  begin
    if IsPrime (I) then
      Tot := Tot + I;
    if I mod (fMax div 100) = 0 then
    begin
      FPosition := I * 100 div fMax;
      Synchronize(UpdateProgress);
    end;
  FTotal := Tot;
  Synchronize(ShowTotal);
end;

procedure TPrimeAdder.ShowTotal;
begin
  ShowMessage ('Thread: ' + IntToStr (FTotal));
end;

procedure TPrimeAdder.UpdateProgress;
begin
  Form1.ProgressBar1.Position := fPosition;
end;
```

The thread object is created when a button is clicked and is automatically destroyed as soon as its `Execute` method is completed:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  AdderThread: TPrimeAdder;
begin
  AdderThread := TPrimeAdder.Create (True);
  AdderThread.Max := Max;
  AdderThread.FreeOnTerminate := True;
  AdderThread.Resume;
end;
```

Instead of setting the maximum number using a property, it would have been better to pass this value as an extra parameter of a custom constructor; I've avoided doing so only to remain focused on the example of using a thread. You'll see more examples of threads in other chapters—particularly Chapter 19, "Internet Programming: Sockets and Indy," which discusses the use of sockets.

Checking for a Previous Instance of an Application

One form of multitasking is the execution of two or more instances of the same application. Any application can generally be executed by a user in more than one instance, and it needs to be able to check for a previous instance already running, in order to disable this default behavior and allow for one instance at most. This section demonstrates several ways of implementing such a check, allowing me to discuss some interesting Windows programming techniques.

Looking for a Copy of the Main Window

To find a copy of the main window of a previous instance, use the `FindWindow` API function and pass it the name of the window class (the name used to register the form's window type, or `WNDCLASS`, in the system) and the caption of the window for which you are looking. In a Delphi application, the name of the `WNDCLASS` window class is the same as the Object Pascal name for the form's class (for example, `TForm1`). The result of the `FindWindow` function is either a handle to the window or zero (if no matching window was found).

The main code of your Delphi application should be written so that it will execute only if the `FindWindow` result is zero:

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', nil);
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
  else
    SetForegroundWindow (Hwnd)
end.
```

To activate the window of the application's previous instance, you can use the `SetForegroundWindow` function, which works for windows owned by other processes. This call produces its effect only if the window passed as parameter hasn't been minimized. When the main form of a Delphi application is minimized, it is hidden, and for this reason the activation code has no effect.

Unfortunately, if you run a program that uses the `FindWindow` call just shown from within the Delphi IDE, a window with that caption and class may already exist: the design-time form. Thus, the program won't start even once. However, it will run if you close the form and its corresponding source code file (closing only the form simply hides the window), or if you close the project and run the program from the Windows Explorer. Consider also that having a form called *Form1* is quite likely to not work as expected, as many Delphi applications might have a form with the same name. This will be fixed in the following versions of the code.

Using a Mutex

A completely different approach is to use a *mutex*, or mutual exclusion object. This is a typical Win32 approach, commonly used for synchronizing threads. Here you will use a mutex to synchronize two different applications or, to be more precise, two instances of the same application.

Once an application has created a mutex with a given name, it can test whether this object is already owned by another application by calling the `WaitForSingleObject` Windows API function. If the mutex has no owner, the application calling this function becomes the owner. If the mutex is already owned, the application waits until the time-out (the function's second parameter) elapses. It then returns an error code.

To implement this technique, you can use the following project source code:

```
var
  hMutex: THandle;
begin
  HMutex := CreateMutex (nil, False, 'OneCopyMutex');
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

If you run this example twice, you'll see that it creates a new, temporary copy of the application (the icon appears in the Taskbar) and then destroys it when the time-out elapses. This approach is certainly more robust than the previous one, but it lacks a feature: How do you enable the existing instance of the application? You still need to find its form, but you can use a better technique.

Searching the Window List

When you want to search for a specific main window in the system, you can use the `EnumWindows` API functions. Enumeration functions are peculiar in Windows, because they usually require another function as a parameter. These enumeration functions require a pointer to a function (often described as a *callback* function) as parameter. This function is applied to each element of the list (in this case, the list of main windows), until the list ends or the function returns False. Here is the enumeration function from the OneCopy example:

```
function EnumWndProc (hwnd: THandle; Param: Cardinal): Bool; stdcall;
var
  ClassName, WinModuleName: string;
  WinInstance: THandle;
begin
  Result := True;
  SetLength (ClassName, 100);
```

```
    GetClassName (hwnd, PChar (ClassName), Length (ClassName));
    ClassName := PChar (ClassName);
    if ClassName = TForm1.ClassName then
    begin
      // get the module name of the target window
      SetLength (WinModuleName, 200);
      WinInstance := GetWindowLong (hwnd, GWL_HINSTANCE);
      GetModuleFileName (WinInstance,
        PChar (WinModuleName), Length (WinModuleName));
      WinModuleName := PChar(WinModuleName); // adjust length
      // compare module names
      if WinModuleName = ModuleName then
      begin
        FoundWnd := Hwnd;
        Result := False; // stop enumeration
      end;
    end;
  end;
```

This function, which is called for each nonchild window of the system, checks the name of each window's class, looking for the name of the `TForm1` class. When it finds a window with this string in its class name, it uses `GetModuleFilename` to extract the name of the executable file of the application that owns the matching form. If the module name matches that of the current program (which was extracted previously with similar code), you can be sure that you have found a previous instance of the same program. Here is how you can call the enumerated function:

```
var
  FoundWnd: THandle;
  ModuleName: string;
begin
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
    ...
  else
  begin
    // get the current module name
    SetLength (ModuleName, 200);
    GetModuleFileName (HInstance, PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar (ModuleName); // adjust length
    // find window of previous instance
    EnumWindows (@EnumWndProc, 0);
```

Handling User-Defined Window Messages

I've mentioned that the `SetForegroundWindow` call doesn't work if the main form of the program has been minimized. Now you can solve this problem. You can ask the form of another application—the previous instance of the same program, in this case—to restore its main form by sending it a user-defined window message. You can then test whether the form is minimized and post a new user-defined message to the old window. Here is the code; in the OneCopy program, it follows the last fragment shown in the preceding section:

```
if FoundWnd <> 0 then
begin
  // show the window, eventually
  if not IsWindowVisible (FoundWnd) then
    PostMessage (FoundWnd, wm_App, 0, 0);
  SetForegroundWindow (FoundWnd);
end;
```

The `PostMessage` API function sends a message to the message queue of the application that owns the destination window. In the form's code, you can add a special function to handle this message:

```
public
  procedure WMApp (var msg: TMessage); message wm_App;

procedure TForm1.WMApp (var msg: TMessage);
begin
  Application.Restore;
end;
```

NOTE *The program uses the* `wm_App` *message rather than the* `wm_User` *message; some system windows use* `wm_User`*, so there is no guarantee that other applications or the system won't send this message. That's why Microsoft introduced* `wm_App` *for messages that are restricted to the application's interpretation.*

Creating MDI Applications

MDI (Multiple Document Interface) is a common approach for an application's structure. An MDI application is made up of several forms that appear inside a single main form. If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application. But with your favorite word processor, you can probably open several different documents, each in its own child window, because the word processor is an MDI application. All these document windows are usually held by a *frame,* or *application,* window.

NOTE *Increasingly, Microsoft is departing from the MDI model stressed in Windows 3 days. Even recent versions of Office tend to use a specific main window for every document: the classic SDI (Single Document Interface) approach. However, MDI isn't dead and can sometimes be a useful structure, as demonstrated by browsers like Opera and Mozilla.*

MDI in Windows: A Technical Overview

The MDI structure gives programmers several benefits automatically. For example, Windows handles a list of the child windows in one of an MDI application's pull-down menus, and specific Delphi methods activate the corresponding MDI functionality to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows:

- The main window of the application acts as a frame or a container.
- A special window, known as the *MDI client,* covers the whole client area of the frame window. This MDI client is one of the Windows predefined controls, just like an edit box or a list box.

The MDI client window lacks any specific user-interface element, but it is visible. You can change the standard system color of the MDI work area (called the Application Background) in the Appearance page of the Display Properties dialog box in Windows.

◆ There are multiple child windows, of the same kind or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the frame window.

Frame and Child Windows in Delphi

Delphi makes it easy to develop MDI applications, even without using the MDI Application template available in Delphi (see the Applications page of the File ➢ New ➢ Other dialog box). You only need to build at least two forms, one with the `FormStyle` property set to fsMDIForm and the other with the same property set to fsMDIChild. Set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at startup, and you need to provide a way to create one or more child windows. You can do so by adding a menu with a New menu item and writing the following code:

```
var
  ChildForm: TChildForm;
begin
  ChildForm := TChildForm.Create (Application);
  ChildForm.Show;
```

Another important feature to add is a Window pull-down menu, which you use as the value of the form's `WindowMenu` property. This pull-down menu will automatically list all the available child windows. (Of course, you can choose any other name for the pull-down menu, but Window is the standard.)

To make this program work properly, you can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  WindowMenu := Window1;
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption := ChildForm.Caption + ' ' + IntToStr (Counter);
  ChildForm.Show;
end;
```

You can also open child windows, minimize or maximize each of them, close them, and use the Window pull-down menu to navigate among them. Now suppose you want to close some of these child windows, to unclutter the client area of your program. Click the Close boxes in some of the child windows, and they are minimized! What is happening? Remember that when you close a window, you generally hide it from view. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, hiding them won't work, because the MDI Window menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi minimizes the MDI child windows when you try to close them. To solve this problem, you need to delete the child windows when they are closed by setting the `Action` reference parameter of the `OnClose` event to caFree.

Building a Complete Window Menu

Your first task is to define a better menu structure for the example. Typically the Window pull-down menu has at least three items: Cascade, Tile, and Arrange Icons. To handle the menu commands, you can use some of the predefined methods of `TForm` that can be used only for MDI frames:

Cascade Method Cascades the open MDI child windows. The windows overlap each other. Iconized child windows are also arranged (see `ArrangeIcons`).

Tile Method Tiles the open MDI child windows; the child forms are arranged so that they do not overlap. The default behavior is horizontal tiling, although if you have several child windows, they will be arranged in several columns. This default can be changed by using the `TileMode` property (set the value to either tbHorizontal or tbVertical).

ArrangeIcons Procedure Arranges all the iconized child windows. Open forms are not moved.

As a better alternative to calling these methods, you can place an ActionList in the form and add to it a series of predefined MDI actions. The related classes are `TWindowArrange`, `TWindowCascade`, `TWindowClose`, `TWindowTileHorizontal`, `TWindowTileVertical`, and `TWindowMinimizeAll`. The connected menu items will perform the corresponding actions and will be disabled if no child window is available. The MdiDemo example, which we'll look at next, demonstrates the use of the MDI actions, among other things.

There are some other interesting methods and properties related strictly to MDI in Delphi:

ActiveMDIChild Property A run-time, read-only property of the MDI frame form that holds the active child window. The user can change this value by selecting a new child window, or the program can change it using the `Next` and `Previous` procedures, which activate the child window following or preceding the currently active one.

ClientHandle Property Holds the Windows handle of the MDI client window, which covers the client area of the main form.

MDIChildren Property An array of child windows you can use together with the `MDIChildCount` property to cycle among all the child windows. This property can be useful for finding a particular child window or to operate on each of them.

Note that the internal order of the child windows is the reverse order of activation. This means the last child window selected is the active window (the first in the internal list), the second-to-last child window selected is the second, and the first child window selected is the last. This order determines how the windows are arranged on the screen. The first window in the list is above all the others, whereas the last window is below all the others, and probably hidden. You can imagine an axis (the z-axis) coming out of the screen toward you. The active window has a higher value for the z coordinate and, thus, covers other windows. For this reason, the Windows ordering schema is known as the *z-order*.

NOTE *The Window menu can be used along with the ActionManager and the ActionMainMenuBar control hosting the menu, starting with Delphi 7. This control has a specific property,* `WindowMenu`*, that you have to set to specify the menu that is going to list the MDI child windows.*

The MdiDemo Example

I've built an example to demonstrate most of the features of a simple MDI application. MdiDemo is a full-blown MDI text editor, because each child window hosts a Memo component and can open and save text files. The child form has a `Modified` property that indicates whether the text of the memo has changed (it is set to True in the handler of the memo's `OnChange` event). `Modified` is set to False in the `Save` and `Load` custom methods and is checked when the form is closed (prompting the user to save the file).

As I've mentioned, the example's main form is based on an ActionList component. The actions are available through some menu items and a toolbar, as shown in Figure 8.3. You can see the details of the ActionList in the example's source code; I'll focus on the code of the custom actions.

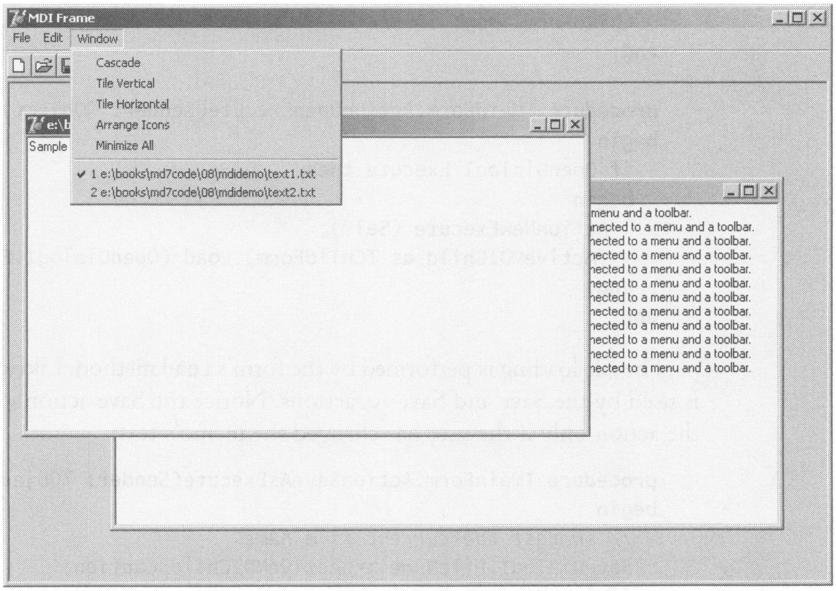

MDI Frame
File
Edit
Window
Cascade
Tile Vertical
Tile Horizontal
Arrange Icons
Minimize All
✓ 1 e:\books\md7code\08\mdidemo\text1.txt
2 e:\books\md7code\08\mdidemo\text2.txt
e:\b
Sample
menu and a toolbar.
nected to a menu and a toolbar.
nected to a menu and a toolbar.

FIGURE 8.3
The MdiDemo program uses a series of predefined Delphi actions connected to a menu and a toolbar.

One of the simplest actions is the `ActionFont` object, which has both an `OnExecute` handler (which uses a FontDialog component) and an `OnUpdate` handler (which disables the action—and hence the associated menu item and toolbar button—when there are no child forms):

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
  if FontDialog1.Execute then
    (ActiveMDIChild as TChildForm).Memo1.Font := FontDialog1.Font;
end;
```

```
procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
  ActionFont.Enabled := MDIChildCount > 0;
end;
```

The action named New creates the child form and sets a default filename. The Open action calls the `ActionNewExcecute` method prior to loading the file:

```
procedure TMainForm.ActionNewExecute(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption :=
    LowerCase (ExtractFilePath (Application.Exename)) + 'text' +
    IntToStr (Counter) + '.txt';
  ChildForm.Show;
end;

procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    ActionNewExecute (Self);
    (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
  end;
end;
```

The file loading is performed by the form's `Load` method. Likewise, the child form's `Save` method is used by the Save and Save As actions. Notice the Save action's `OnUpdate` handler, which enables the action only if the user has changed the memo's text:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
begin
  // suggest the current file name
  SaveDialog1.FileName := ActiveMDIChild.Caption;
  if SaveDialog1.Execute then
  begin
    // modify the file name and save
    ActiveMDIChild.Caption := SaveDialog1.FileName;
    (ActiveMDIChild as TChildForm).Save;
  end;
end;

procedure TMainForm.ActionSaveUpdate(Sender: TObject);
begin
  ActionSave.Enabled := (MDIChildCount > 0) and
    (ActiveMDIChild as TChildForm).Modified;
end;
```

```
procedure TMainForm.ActionSaveExecute(Sender: TObject);
begin
  (ActiveMDIChild as TChildForm).Save;
end;
```

MDI Applications with Different Child Windows

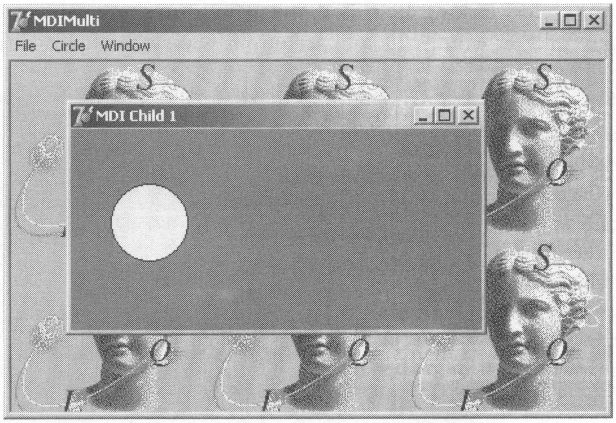
In complex MDI applications, it's common to include child windows of different kinds (that is, based on different child forms). I built an example called MdiMulti to highlight some problems you may encounter with this approach. This example has two different types of child forms: the first type hosts a circle drawn in the position of the last mouse click, and the second contains a bouncing square. The main form also has a custom background, obtained by painting a tiled image in it.

Child Forms and Merging Menus

The first type of child form displays a circle in the position where the user clicked one of the mouse buttons. Figure 8.4 shows an example of the output of the MdiMulti program. The program includes a Circle menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. It's interesting that to program the child form, you do not need to consider the existence of other forms or of the frame window. You simply write the code for the form, and that's all. The only special care required is for the menus of the two forms.

FIGURE 8.4

The output of the MdiMulti example, with a child window that displays circles



If you prepare a main menu for the child form, it will replace the main menu of the frame window when the child form is activated: An MDI child window cannot have a menu of its own. But the fact that a child window can't have any menus should not bother you, because this is the standard behavior of MDI applications. You can use the frame window's menu bar to display the child window's menus. Even better, you can merge the frame window's menu bar with that of the child form. For example, in this program, the child form's menu can be placed between the frame window's File and Window pull-down menus. You can accomplish this using the following `GroupIndex` values:

- File pull-down menu, main form: 1

- Circle pull-down menu, child form: 2
- Window pull-down menu, main form: 3

Using these settings for the menu group indexes, the menu bar of the frame window will have either two or three pull-down menus. At startup, the menu bar has two menus. As soon as you create a child window, there are three menus; and when the last child window is closed (destroyed), the Circle pull-down menu disappears. You should spend some time testing this behavior by running the program.

The second type of child form shows a moving image. The square (a Shape component) moves around the client area of the form at fixed time intervals, using a Timer component, and bounces against the edges of the form, changing its direction. This turning process is determined by a fairly complex algorithm, which I don't have space to examine; the main point of the example is to show you how menu merging behaves when you have an MDI frame with child forms of different types. (You can study the source code to see how it works.)

The Main Form

Now let's integrate the two child forms into an MDI application. The File pull-down menu has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. The Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the main form's original menu bar is reset. By using the proper menu group indexes, you let Delphi accomplish everything automatically, as you can see in Figure 8.5.

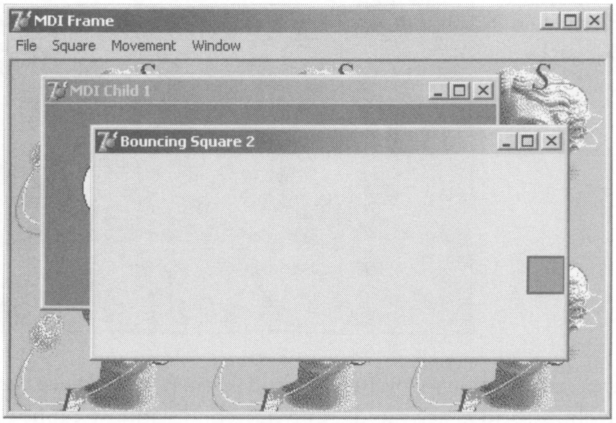


FIGURE 8.5 The menu bar of the MdiMulti application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of Figure 8.4.

I've added a few other menu items in the main form to close every child window and show some statistics about them. The method related to the `Count` command scans the `MDIChildren` array property to count the number of child windows of each kind (using the RTTI operator `is`):

```
for I := 0 to MDIChildCount - 1 do
  if MDIChildren is TBounceChildForm then
```

```
    Inc (NBounce)
  else
    Inc (NCircle);
```

Subclassing the MDI Client Window

The example program also includes support for a background-tiled image. The bitmap is taken from an Image component and should be painted on the form in the `wm_EraseBkgnd` Windows message's handler. The problem is that you cannot simply connect the code to the main form, because a separate window (the MDI Client) covers its surface.

You have no corresponding Delphi form for this window, so how can you handle its messages? You have to resort to a low-level Windows programming technique known as *subclassing*. (In spite of the name, it has little to do with OOP inheritance.) The basic idea is that you can replace the window procedure that receives all the window messages with a new procedure you provide. You can do so by calling the `SetWindowLong` API function and providing the memory address of the procedure (the function pointer).

NOTE *A window procedure is a function that receives all the messages for a window. Every window must have a window procedure and can have only one. Even Delphi forms have a window procedure; although it is hidden in the system, it calls the* `WndProc` *virtual function, which you can use. However, the VCL has a predefined handler for the messages, which are then forwarded to the form's message-handling methods after some preprocessing. With all this support, you need to handle window procedures explicitly only when working with non-Delphi windows, as in this case.*

Unless you have a reason to change the default behavior of this system window, you can simply store the original procedure and call it to obtain default processing. The two function pointers referring to the two procedures (old and new) are stored in two local fields on the form:

```
private
  OldWinProc, NewWinProc: Pointer;
  procedure NewWinProcedure (var Msg: TMessage);
```

The form also has a method you'll use as a new window procedure; the code will be used to paint on the background of the window. Because this is a method and not a plain window procedure, the program has to call the `MakeObjectInstance` method to add a prefix to the method and let the system use it as if it were a function. All this description is summarized by just two complex statements:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  NewWinProc := MakeObjectInstance (NewWinProcedure);
  OldWinProc := Pointer (SetWindowLong (ClientHandle, gwl_WndProc, Cardinal
    (NewWinProc)));
  OutCanvas := TCanvas.Create;
end;
```

The window procedure you install calls the default procedure. Then, if the message is `wm_EraseBkgnd` and the image is not empty, you draw it on the screen many times using the `Draw` method of a temporary canvas. This canvas object is created when the program starts (see the previous code) and connected to the handle passed as `wParam` parameter by the message. With this approach, you don't have to create a new `TCanvas` object for every background painting operation requested, thus saving a little time in the frequent operation. Here is the code, which produces the output already seen in Figure 8.5:

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
  BmpWidth, BmpHeight: Integer;
  I, J: Integer;
begin
  // default processing first
  Msg.Result := CallWindowProc (OldWinProc, ClientHandle, Msg.Msg, Msg.wParam,
    Msg.lParam);

  // handle background repaint
  if Msg.Msg = wm_EraseBkgnd then
  begin
    BmpWidth := MainForm.Image1.Width;
    BmpHeight := MainForm.Image1.Height;
    if (BmpWidth <> 0) and (BmpHeight <> 0) then
    begin
      OutCanvas.Handle := Msg.wParam;
      for I := 0 to MainForm.ClientWidth div BmpWidth do
        for J := 0 to MainForm.ClientHeight div BmpHeight do
          OutCanvas.Draw (I * BmpWidth, J * BmpHeight,
            MainForm.Image1.Picture.Graphic);
    end;
  end;
end;
```

Visual Form Inheritance

When you need to build two or more similar forms, possibly with different event handlers, you can use dynamic techniques, hide or create new components at run time, change event handlers, and use `if` or `case` statements. Or, you can apply the object-oriented techniques, thanks to visual form inheritance. In short, instead of creating a form based on `TForm`, you can inherit a form from an existing form, adding new components or altering the properties of the existing components. But what is the advantage of visual form inheritance?

It mostly depends on the kind of application you are building. If the program has multiple forms, some of which are very similar or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the application's forms.

You can also use visual form inheritance to customize an application for different clients without duplicating any source code or form definition code—you inherit the specific versions for a client from the standard forms. Remember, the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method in a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

NOTE *Delphi includes another feature that resembles visual form inheritance: frames. In both cases, you can work at design time on two versions of a form/frame. However, in visual form inheritance, you define two different classes (parent and derived), whereas with frames, you work on a class and an instance. Frames are discussed in detail later in this chapter.*

Inheriting from a Base Form

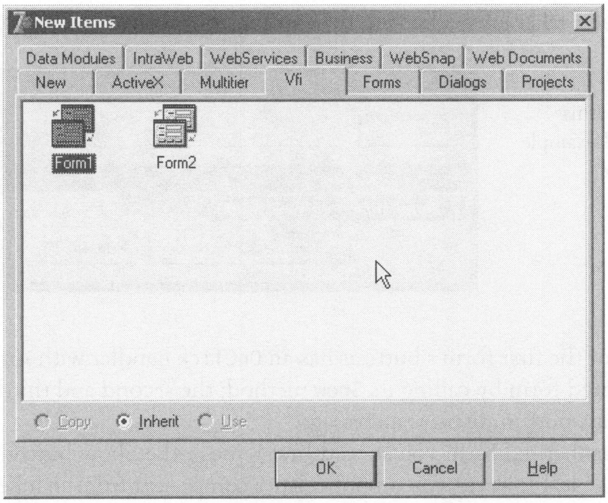
The rules governing visual form inheritance are simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command in the Object Inspector. You can do the same thing by setting the two properties to the same value and recompiling the code. After modifying multiple properties, you can resynchronize them all to the base version by applying the Revert to Inherited command from the component's local menu.

Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To describe how visual form inheritance works, I've built a simple example called VFI. To build it, first start a new project and add four buttons to its main form. Then select File ➢ New ➢ Other and choose the page with the name of the project in the New Items dialog box (see Figure 8.6).

FIGURE 8.6
The New Items dialog box allows you to create an inherited form.



○ Copy ● Inherit ○ Use

In the New Items dialog, you can choose the form from which you want to inherit. The new form has the same four buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

And here is its initial class declaration, where you can see that the base class is not the usual `TForm` but the base class form:

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

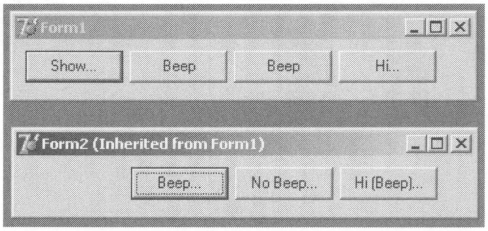
Notice the presence of the `inherited` keyword in the textual description; also notice that the form has some components, although they are defined in the base class form. If you move the form and add the caption of one of the buttons, the textual description changes accordingly:

```
inherited Form2: TForm2
  Left = 313
  Top = 202
  Caption = 'Form2'
  inherited Button2: TButton
    Caption = 'Beep...'
  end
end
```

Only the properties with a different value are listed (and by removing these properties from the textual description of the inherited form, you can reset them to the value of the base form, as I mentioned earlier). I've changed the captions of most of the buttons, as you can see in Figure 8.7.

FIGURE 8.7

The two forms of the VFI example at run time



Each of the first form's buttons has an `OnClick` handler with simple code. The first button shows the inherited form by calling its `Show` method; the second and third buttons call the `Beep` procedure; and the last button displays a message.

In the inherited form you should first remove the Show button, because the secondary form is already visible. However, you cannot delete a component from an inherited form. An alternative solution is to set the component's `Visible` property to False—the button will still be there, but it won't be visible (as you can guess from Figure 8.7). The other three buttons will be visible but with different handlers. If you select the `OnClick` event of a button in the inherited form (by double-clicking it), you'll get an empty method that's slightly different from the default one, because it includes the `inherited` keyword. This keyword stands for a call to the corresponding event handler of the base form. Notice, though, that this keyword is always added by Delphi, even if the handler is not defined in the parent class (and this is reasonable, because it might be defined later) or if the component is not present in the parent class (which doesn't seem like a great idea to me). It is simple to execute the base form's code and perform some other operations:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hi');
end;
```

This is not the only choice. Alternatively, you can write a new event handler and not execute the base class's code, as I've done for the VFI example's third button: To accomplish this, simply remove the `inherited` keyword.

Still another choice includes calling a base-class method after some custom code has been executed, calling it when a condition is met, or calling the handler of a different event of the base class, as I've done for the fourth button:

```
procedure TForm2.Button4Click(Sender: TObject);
begin
  inherited Button3Click (Sender);
  inherited;
end;
```

You probably won't inherit from a different handler often, but you must be aware that you can. Of course, you can consider each method of the base form as a method of your form, and call it freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at real-world examples more complex than this book has room to explore. Next I want to show you *visual form polymorphism*.

NOTE *Visual form inheritance doesn't work nicely with collections: You cannot extend a collection property of a component in an inherited form. This limitation prevents the practical use of a series of components like Toolbars or ListViews with details. Of course, you can use those components in the parent or inherited form, but you cannot extend the elements they contain, because they are stored in a collection. A solution to this problem is to avoid assigning these collections at design time, and instead use a run-time technique. You'll still use form inheritance, but lose the* visual *portion of it. If you try to use the Action Manager component, you'll find you cannot even inherit from a form hosting it. Borland disabled this feature, because it would cause you too much trouble.*

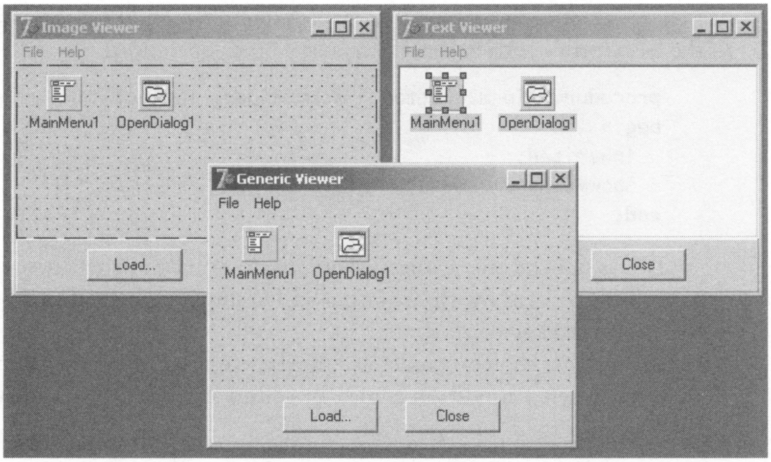
Polymorphic Forms

If you add an event handler to a form and then change it in an inherited form, there is no way to refer to the two methods using a common variable of the base class, because the event handlers use static binding by default.

Confusing? Here is an example, which is intended for experienced Delphi programmers. Suppose you want to build a bitmap viewer form and a text viewer form in the same program. The two forms have similar elements, a similar toolbar, a similar menu, an OpenDialog component, and different components for viewing the data. So, you decide to build a base-class form containing the common elements and inherit the two forms from it. You can see the three forms at design time in Figure 8.8.

FIGURE 8.8

The base-class form and the two inherited forms of the PoliForm example at design time



The main form contains a toolbar panel with a few buttons (real toolbars have a few problems with visual form inheritance), a menu, and an open dialog component. The two inherited forms have only minor differences, but they feature a new component: either an image viewer (`TImage`) or a text viewer (`TMemo`). They also modify the settings of the OpenDialog component, to refer to different types of files.

The main form includes some common code. The Close button and the File ➢ Close command call the `Close` method of the form. The Help ➢ About command shows a simple message box. The base form's Load button has only a `ShowMessage` call displaying an error message. The File ➢ Load command calls another method:

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
  LoadFile;
end;
```

This method is defined in the `TViewerForm` class as a virtual abstract method (so that the class of the base form is an abstract class). Because this is an abstract method, you must redefine it (and override it) in the inherited forms. The code for this `LoadFile` method uses the `OpenDialog1` component to ask the user to select an input file and loads it into the image component:

```
procedure TImageViewerForm.LoadFile;
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (OpenDialog1.Filename);
end;
```

The other inherited class has similar code, which loads the text into the memo component. The project has one more form, a main form with two buttons, that reloads the files in each of the viewer forms. The main form is the only form created by the project when it starts. The generic viewer form is never created: It is only a generic base class, containing common code and components of the two subclasses. The forms of the two subclasses are created in the main form's `OnCreate` event handler:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  FormList [1] := TTextViewerForm.Create (Application);
  FormList [2] := TImageViewerForm.Create (Application);
  for I := 1 to 2 do
    FormList[I].Show;
end;
```

`FormList` is a *polymorphic* array of generic `TViewerForm` objects, declared in the `TMainForm` class. Note that to make this declaration in the class, you need to add the Viewer unit (but not the specific forms) in the **uses** clause of the interface portion of the main form. The array of forms is used to load a new file in each viewer form when one of the two buttons is clicked. The handlers of the two buttons' `OnClick` events use different approaches:

```
// ReloadButton1Click
for I := 1 to 2 do
  FormList [I].ButtonLoadClick (Self);

// ReloadButton2Click
for I := 1 to 2 do
  FormList [I].LoadFile;
```

The second button calls a virtual method, and it works without any problem. The first button calls an event handler and always reaches the generic `TFormView` class (displaying the error message of its `ButtonLoadClick` method). This happens because the method is static, not virtual.

To make this approach work, you can declare the `ButtonLoadClick` method of the `TFormView` class as virtual and declare it as overridden in each of the inherited form classes, as you do for any other virtual method:

```
type
  TViewerForm = class(TForm)
    procedure ButtonLoadClick(Sender: TObject); virtual;
  public
    procedure LoadFile; virtual; abstract;
  end;

type
  TImageViewerForm = class(TViewerForm)
    procedure ButtonLoadClick(Sender: TObject); override;
```

```
public
  procedure LoadFile; override;
end;
```

This trick really works, although it is never mentioned in the Delphi documentation. This ability to use virtual event handlers is what I mean by visual form polymorphism. In other (more technical) words, you can assign a virtual method to an event property, which will take the address of the method according to the instance available at run time.

Understanding Frames

Chapter 1, "Delphi 7 And Its IDE," briefly discussed frames. You've seen that you can create a new frame, place components in it, write event handlers for the components, and then add the frame to a form. In other words, a frame is similar to a form, but it defines only a portion of a window, not a complete window. The interesting element of frames is that you can create multiple instances of a frame at design time, and you can modify the class and the instance at the same time. Thus frames are an effective tool for creating customizable composite controls at design time—something close to a visual component-building tool.

In visual form inheritance, you can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one (unless doing so overrides a property or event). With frames, you work on a class (as usual in Delphi), but you can also customize one or more instances of the class at design time. When you work on a form, you cannot change a property of the `TForm1` class for a specific instance of this form, and not the others, at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, you can customize each instance at design time. You could already do this with component templates, but component templates were based on the concept of copying and pasting components and their code. You could not change the original definition of the template and see the effect every place it was used. With frames (and, in a different way, with visual form inheritance), changes to the original version (the class) are reflected in the copies (the instances).

Let's discuss a few more elements of frames with an example called Frames2. This program has a frame with a list box, an edit box, and three buttons with code operating on the components. The frame also has a bevel aligned to its client area, because frames have no border. Of course, the frame has also a corresponding class, which looks like a form class:

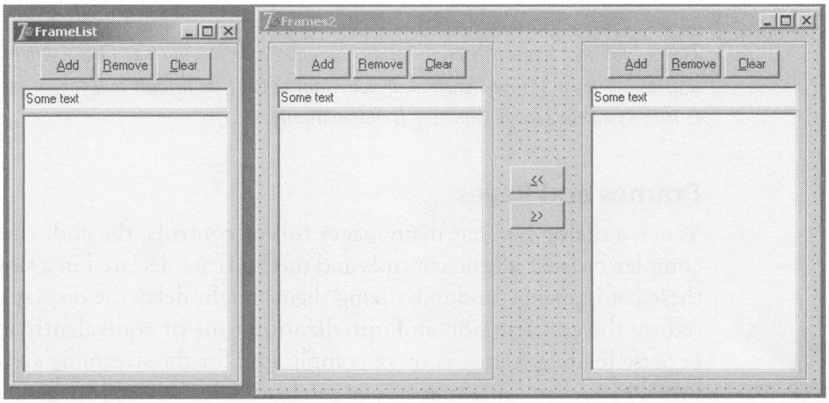
```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

What is different from a form is that you can add the frame to a form. I've used two instances of the frame in the example (as you can see in Figure 8.9) and modified the behavior slightly. The first instance of the frame has the list box items sorted. When you change a property of a component of a frame, the DFM file of the hosting form will list the differences, as it does with visual form inheritance:

```
object FormFrames: TFormFrames
  Caption = 'Frames2'
  inline FrameList1: TFrameList
    Left = 8
    Top = 8
    inherited ListBox: TListBox
      Sorted = True
    end
  end
  inline FrameList2: TFrameList
    Left = 232
    Top = 8
    inherited btnClear: TButton
      OnClick = FrameList2btnClearClick
    end
  end
end
```

FIGURE 8.9

A frame and two instances of it at design time, in the Frames2 example



As you can see from the listing, the DFM file for a form that has frames uses a specific DFM keyword, `inline`. The references to the modified components of the frame, however, use the `inherited` keyword, although this term is used with an extended meaning: In this case, `inherited` doesn't refer to a base class you are inheriting from, but to the class from which you are instancing (or inheriting) an object. It was a good idea, though, to use an existing feature of visual form inheritance and apply it to the new context. This approach lets you use the Revert to Inherited command of the Object Inspector or of the form to cancel the changes and get back to the default value of properties.

Notice also that unmodified components of the frame class are not listed in the DFM file of the form using the frame; and, the form has two frames with different names, but the components on the two frames have the same name. These components are not owned by the form, but are owned by the frame. This implies that the form has to reference those components through the frame, as you can see in the code for the buttons that copy items from one list box to the other:

```
procedure TFormFrames.btnLeftClick(Sender: TObject);
begin
  FrameList1.ListBox.Items.AddStrings (FrameList2.ListBox.Items);
end;
```

Finally, in addition to modifying properties of any instance of a frame, you can change the code of any of its event handlers. If you double-click one of the frame's buttons while working on the form (not on the stand-alone frame), Delphi will generate this code for you:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  FrameList2.btnClearClick(Sender);
end;
```

The line of code automatically added by Delphi corresponds to a call to the inherited event handler of the base class in visual form inheritance. This time, however, to get the default behavior of the frame, you need to call an event handler and apply it to a specific instance—the frame object itself. The current form doesn't include this event handler and knows nothing about it. Whether you leave this call in place or remove it depends on the effect you are looking for.

TIP *Note that because the event handler has some code, leaving it as Delphi generated it and saving the form won't remove it as usual: It isn't empty! Instead, if you want to omit the default code for an event, you need to add at least a comment to it to avoid the system removing it automatically.*

Frames and Pages

When a dialog box has many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared in a single form. In addition, creating all these components (and initializing them) might delay the display of the dialog box. Frames don't reduce the construction and initialization time of equivalently loaded forms; quite the contrary, because loading frames is more complicated for the streaming system than loading simple components. However, using frames, you can load only the visible pages of a multipage dialog box, reducing the *initial* load time, which is what the user perceives.

Frames can solve both of these issues. You can easily divide the code of a single complex form into one frame per page. The form will host all the frames in a PageControl. This approach yields simpler, more focused units and makes it easier to reuse a specific page in a different dialog box or application. Reusing a single page of a PageControl without using a frame or an embedded form is far from simple. (For an alternative approach, see the sidebar "Forms in Pages.")

As an example of this approach, I've built the FramePag example. It has some frames placed inside the three pages of a PageControl, as you can see in Figure 8.10 by looking at the Object TreeView on the side of the design-time form. All the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them. Two of the pages have the same frame, but the two instances of the frame have some differences at design time. The `Frame3` frame in the example has a list box populated with a text file at startup, and it has buttons to modify the items in the list and save them to a file. The filename is placed in a label so you can easily select a file for the frame at design time by changing the `Caption` of the label.

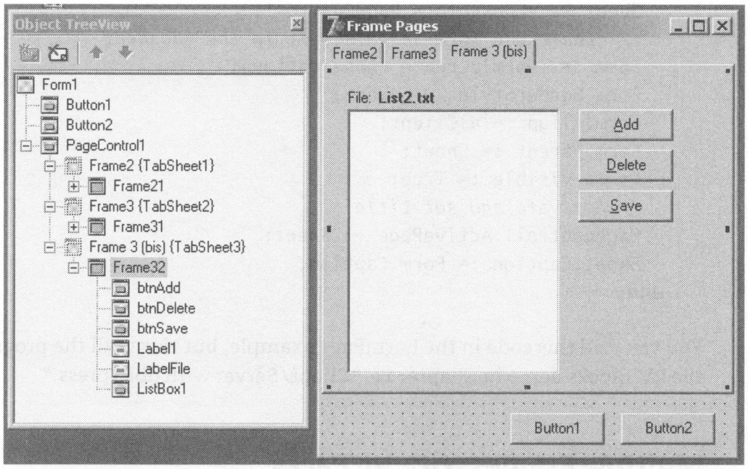


FIGURE 8.10
Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks.

Being able to use multiple instances of a frame is one of the reasons this technique was introduced, and customizing the frame at design time is even more important. Because adding properties to a frame and making them available at design time requires some customized and complex code, it is nice to use a component to host these custom values. You have the option of hiding these components (such as the label in this example) if they don't pertain to the user interface.

In the example, you need to load the file when the frame instance is created. Because frames have no `OnCreate` event, your best choice is probably to override the `CreateWnd` method. Writing a custom constructor doesn't work, because it is executed too early—before the specific label text is available. In the `CreateWnd` method, you load the list box content from a file.

NOTE *When questioned about the issue of the missing* `OnCreate` *event handler for frames, Borland R&D members have stated that they could not fire it in correspondence with the* `wm_Create` *message, because it happens with forms. The creation of the frame window (as is true for most controls) is delayed for performance reasons. More trouble happens in the case of inheritance among forms holding frames, so to avoid problems, this features has been disabled—programmers can write the code they deem reasonable.*

FORMS IN PAGES

Although you can use frames to define the pages of a PageControl at design time, I generally use other forms at run time. This approach leaves me with the flexibility of having the pages defined in separate units (and DFM files) but at the same time allows me to also use those forms as stand-alone windows. In addition, I avoid having to live with the subtly different behaviors of frames.

Once you have a main form with a page control and one or more secondary forms to display in it, all you have to do is write the following code to create the secondary forms and place them in the pages:

```
var
  Form: TForm;
  Sheet: TTabSheet;
begin
  // create a tabsheet within the page control
  Sheet := TTabSheet.Create(PageControl1);
  Sheet.PageControl := PageControl1;
  // create the form and place it in the tabsheet
  Form := TForm2.Create (Application);
  Form.BorderStyle := bsNone;
  Form.Align := alClient;
  Form.Parent := Sheet;
  Form. Visible := True;
  // activate and set title
  PageControl1.ActivePage := Sheet;
  Sheet.Caption := Form.Caption;
end;
```

You can find this code in the FormPage example, but this is all the program does. For an application, see the RWBlocks demo in Chapter 14, "Client/Server with dbExpress."

Multiple Frames with No Pages

Another approach avoids creating all the pages along with the form hosting them, by leaving the PageControl empty and creating the frames only when a page is displayed. When you have frames on multiple pages of a PageControl, the windows for the frames are created only when they are first displayed, as you can find out by placing a breakpoint in the creation code of the previous example.

As an even more radical approach, you can get rid of the page controls and use a TabControl. Used this way, the tab has no connected tab sheets (or pages) and can display only one set of information at a time. For this reason, you must create the current frame and either destroy the previous one or hide it by setting its `Visible` property to False or calling the new frame's `BringToFront`. Although this sounds like a lot of work, in a large application this technique can be worth it because of the reduced resource and memory usage you can obtain.

To demonstrate this approach, I've built the FrameTab example, which is similar to the previous one but it is based on a TabControl and dynamically created frames. The main form, visible at run time in Figure 8.11, has a TabControl with one page for each frame:

```
object Form1: TForm1
  Caption = 'Frame Pages'
```

```
    OnCreate = FormCreate
    object Button1: TButton...
    object Button2: TButton...
    object Tab: TTabControl
      Anchors = [akLeft, akTop, akRight, akBottom]
      Tabs.Strings = ( 'Frame2' 'Frame3' )
      OnChange = TabChange
    end
  end
```

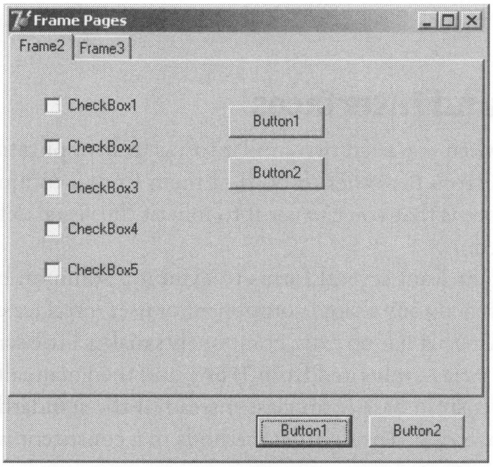


FIGURE 8.11

The first page of the FrameTab example at run time. The frame inside the tab is created at run time.

I've given each tab a caption corresponding to the name of the frame, because I'll use this information to create the new pages. When the form is created, and whenever the user changes the active tab, the program gets the tab's current caption and passes it to the custom `ShowFrame` method. The method's code checks whether the requested frame already exists (frame names in this example follow the Delphi standard of having a number appended to the class name) and then brings it to the front. If the frame doesn't exist, the method uses the frame name to find the related frame class, creates an object of that class, and assigns a few properties to it. The code makes extensive use of class references and dynamic creation techniques:

```
type
  TFrameClass = class of TFrame;

procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + '1') as TFrame;
  if not Assigned (Frame) then
```

```
    begin
      FrameClass := TFrameClass (FindClass ('T' + FrameName));
      Frame := FrameClass.Create (Self);
      Frame.Parent := Tab;
      Frame.Visible := True;
      Frame.Name := FrameName + '1';
    end;
    Frame.BringToFront;
  end;
```

To make this code work, remember to add a call to `RegisterClass` in the initialization section of each unit defining a frame.

Base Forms and Interfaces

You have seen that when you need two similar forms in an application, you can use visual form inheritance to inherit one from the other or both of them from a common ancestor. The advantage of visual form inheritance is that you can use it to inherit the visual definition: the DFM. However, this is not always required.

At times, you might want several forms to exhibit a common behavior or respond to the same commands without having any shared component or user interface elements. Using visual form inheritance with a base form that has no extra components makes little sense to me. I rather prefer to define my own custom form class, inherited from `TForm`, and then manually edit the form class declarations to inherit from this custom base form class instead of the standard one. If you only need to define shared methods or override `TForm` virtual methods in a consistent way, defining custom form classes can be a good idea.

Using a Base Form Class

A simple demonstration of this technique is available in the FormIntf demo; it also showcases the use of interfaces for forms. In a new unit called SaveStatusForm, I've defined the following form class (with no related DFM file—instead of using the New Form command, create a new unit and type the code in it):

```
type
  TSaveStatusForm = class (TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

The two overridden methods are called at the same time as the event handler so you can attach extra code (allowing the event handler to be defined as usual). Inside the two methods, you load or save the form position in an INI file of the application, in a section marked with the form caption. Here is the code for the two methods:

```
procedure TSaveStatusForm.DoCreate;
var
  Ini: TIniFile;
```

```
begin
  inherited;
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Left := Ini.ReadInteger(Caption, 'Left', Left);
  Top := Ini.ReadInteger(Caption, 'Top', Top);
  Width := Ini.ReadInteger(Caption, 'Width', Width);
  Height := Ini.ReadInteger(Caption, 'Height', Height);
  Ini.Free;
end;

procedure TSaveStatusForm.DoDestroy;
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Ini.WriteInteger(Caption, 'Left', Left);
  Ini.WriteInteger(Caption, 'Top', Top);
  Ini.WriteInteger(Caption, 'Width', Width);
  Ini.WriteInteger(Caption, 'Height', Height);
  Ini.Free;
  inherited;
end;
```

Again, this is a simple common behavior for your forms, but you can define a complex class here. To use this as a base class for the forms you create, let Delphi create the forms as usual (with no inheritance) and then update the form declaration to something like the following:

```
type
  TFormBitmap = class(TSaveStatusForm)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    ...
```

Simple as it seems, this technique is very powerful, because all you need to do is change the definition of your application's forms to refer to this base class. If even this step is too tedious, because you might want to change this base class in your program at some point, you can use an extra trick: "interposer" classes.

INI FILES AND THE REGISTRY IN DELPHI

To save information about the status of an application in order to restore it the next time the program is executed, you can use the explicit support Windows provides for storing this kind of information. INI files, the old Windows standard, are once again the preferred way to save application data. The alternative is the Registry, which is still quite popular. Delphi provides ready-to-use classes to manipulate both.

THE *TINIFILE* CLASS

For INI files, Delphi has a `TIniFile` class. Once you have created an object of this class and connected it to a file, you can read and write information to it. To create the object, you need to call the constructor,

Continued on next page

INI FILES AND THE REGISTRY IN DELPHI *(continued)*

passing a filename to it, as in the following code:

```
var
  IniFile: TIniFile;
 begin
   IniFile := TIniFile.Create ('myprogram.ini');
```

There are two choices for the location of the INI file. The code just listed will store the file in the Windows directory or a user folder for settings in Windows 2000. To store data locally to the application (as opposed to local to the current user), you should provide a full path to the constructor.

INI files are divided into sections, each indicated by a name enclosed in square brackets. Each section can contain multiple items of three possible kinds: strings, integers, or Booleans. The `TIniFile` class has three Read methods, one for each kind of data: `ReadBool`, `ReadInteger`, and `ReadString`. There are also three corresponding methods to write the data: `WriteBool`, `WriteInteger`, and `WriteString`. Other methods allow you to read or erase a whole section. In the Read methods, you can also specify a default value to be used if the corresponding entry doesn't exist in the INI file.

Notice that Delphi uses INI files quite often, but they are disguised with different names. For example, the desktop (`.dsk`) and options (`.dof`) files are structured as INI files.

THE *TREGISTRY* AND *TREGINIFILE* CLASSES

The Registry is a hierarchical database of information about the computer, software configuration, and user preferences. Windows has a set of API functions to interact with the Registry; you basically open a key (or folder) and then work with subkeys (or subfolders) and values (or items), but you must be aware of the structure and the details of the Registry.

Delphi provides two approaches to using the Registry. The `TRegistry` class provides a generic encapsulation of the Registry API, whereas the `TRegIniFile` class provides the interface of the `TIniFile` class but saves the data in the Registry. This class is the natural choice for portability between INI-based and Registry-based versions of the same program. When you create a `TRegIniFile` object, your data ends up in the current user information, so you'll generally use a constructor like this:

```
IniFile := TRegIniFile.Create ('Software\MyCompany\MyProgram');
```

By using the `TIniFile` and the `TRegIniFile` classes offered by the VCL, you can move from one model of local and per-user storage to the other. Not that I think you should use the Registry much, because having a centralized repository for the settings of each application was an architectural error—even Microsoft acknowledges this fact (without really admitting the error) by suggesting, in the Windows 2000 Compatibility Requirements, that you no longer use the Registry for applications settings, but instead go back to using INI files within the Documents and Settings folder of the current user (something not many programmers know of).

AN EXTRA TRICK: INTERPOSER CLASSES

In contrast with Delphi VCL components, which must have unique names, Delphi classes in general must be unique only within their unit. Thus you can have two different units defining a class with the same name. This technique looks weird at first sight, but can be useful. For example, Borland uses this approach to provide compatibility between VCL and VisualCLX classes. Both have a `TForm` class, one defined in the Forms unit and the other in the QForms unit.

NOTE *This technique is much older than CLX/VCL. For example, the service and control panel applet units define their own* `TApplication` *object, which is not related to the* `TApplication` *used by VCL visual GUI applications and defined in the Forms unit.*

I've seen a technique called "interposer classes" mentioned in an old issue of *The Delphi Magazine*. It suggested replacing standard Delphi class names with your own versions that have the same class name. This way, you can use Delphi's form designer and refer to Delphi standard components at design time, but use your own classes at run time.

The idea is simple. In the SaveStatusForm unit, you can define the new form class as follows:

```
type
  TForm = class (Forms.TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;
```

This class is called `TForm`, and it inherits from `TForm` of the Forms unit (this last reference is compulsory to avoid a kind of recursive definition). In the rest of the program, you don't need to change the class definition for your form, but simply add the unit defining the interposer class (the SaveStatusForm unit in this case) in the `uses` statement *after* the unit defining the Delphi class. The order of the unit in the `uses` statement is important, and is the reason some people criticize this technique, because it is difficult to know what is going on. I have to agree: I find interposer classes handy at times (more for components than for forms), but their use makes programs less readable and at times harder to debug.

Using Interfaces

Another technique, which is slightly more complex but even more powerful than the definition of a common base form class, is to create forms that implement specific interfaces. You can have forms that implement one or more of these interfaces, query each form for the interfaces it implements, and call the supported methods.

As an example (available in the same FormIntf program I began discussing in the last section), I've defined a simple interface for loading and storing:

```
type
  IFormOperations = interface
    ['{DACFDB76-0703-4A40-A951-10D140B4A2A0}']
    procedure Load;
    procedure Save;
  end;
```

Each form can optionally implement this interface, as in the following `TFormBitmap` class:

```
type
  TFormBitmap = class(TForm, IFormOperations)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    SavePictureDialog1: TSavePictureDialog;
  public
    procedure Load;
    procedure Save;
  end;
```

The example code includes the `Load` and `Save` methods, which use the standard dialog boxes to load or save the image. (In the example's code, the form also inherits from the `TSaveStatusForm` class.)

When an application has one or more forms implementing interfaces, you can apply a given interface method to all the forms supporting it, with code like this (extracted from the main form of the FormIntf example):

```
procedure TFormMain.btnLoadClick(Sender: TObject);
var
  i: Integer;
  iFormOp: IFormOperations;
begin
  for i := 0 to Screen.FormCount - 1 do
    if Supports (Screen.Forms [i], IFormOperations, iFormOp) then
      iFormOp.Load;
end;
```

Consider a business application in which you can synchronize all the forms to the data of a specific company or a specific business event. Also consider that, unlike inheritance, you can have several forms that implement multiple interfaces, with unlimited combinations. This is why using such an architecture can improve a complex Delphi application a great deal, making it much more flexible and easier to adapt to implementation changes.

Delphi's Memory Manager

I'll end this chapter devoted to the structure of Delphi applications with a section devoted to memory management. This topic is very complex, and probably worth an entire chapter of its own; here I can only scratch it and provide a few indications for further experiments. For more detailed memory analysis you can refer to the many Delphi add-on tools addressing memory verification and control, including MemCheck, MemProof, MemorySleuth, Code Watch, and AQTime.

Delphi has a memory manager, accessible using the `GetMemoryManager` and `SetMemoryManager` functions of the System unit. These functions allow you to retrieve the current memory manager record or modify it with your custom memory manager. A *memory manager record* is a set of three functions used to allocate, deallocate, and reallocate memory:

```
type
  TMemoryManager = record
    GetMem: function(Size: Integer): Pointer;
    FreeMem: function(P: Pointer): Integer;
    ReallocMem: function(P: Pointer; Size: Integer): Pointer;
  end;
```

It's important to know how these functions are called when you create an object, because you can hook in two different steps. As you call a constructor, Delphi invokes the `NewInstance` virtual class function, defined in `TObject`. Because this is a virtual function, you can modify the memory manager for a specific class by overriding it. To perform the memory allocation, however, `NewInstance` typically ends up calling the `GetMem` function of the active memory manager, which provides you with your second chance to customize the standard behavior.

Unless you have very special needs, you won't generally need to hook into the memory manager to modify how memory allocation works. However, I find it quite useful to hook into a memory manager to determine whether memory allocation is working properly—that is, to be sure the program has no memory leaks. For example, you can override a class's `NewInstance` and `FreeInstance` methods to keep a count of the number of objects of the class being created and destroyed and check if the total is zero.

An even simpler technique is to perform the same test over the number of objects allocated by the entire memory manager. In the early versions of Delphi doing so required extra code, but the memory manager exposes two global variables (`AllocMemCount` and `AllocMemSize`) that can help you determine what is going on in the system.

NOTE *For more detailed information about how the memory manager is working internally, you can use the* `GetHeapStatus` *function. It is available only on Windows, because it provides information about the status of the memory allocator. On Linux, the RTL uses the system allocator, not a custom memory allocator.*

The simplest way to determine whether your program is handling memory properly is to test whether `AllocMemCount` goes back to zero. The problem is deciding when to perform such a test. A program begins by executing the initialization section of its units, which usually allocate memory freed by the respective finalization sections. To guarantee that your code is executed at the very end, you must write it in the finalization section of a unit and place it at the very beginning of the units list in the project source code file. You can see such a unit in Listing 8.1. This is the SimpleMemTest unit of the ObjsLeft example, which has a sample form with a button to show the current allocations count and a button to create a memory leak (which is then caught when the program terminates).

LISTING 8.1: A SIMPLE UNIT FOR TESTING MEMORY LEAKS, FROM THE OBJSLEFT EXAMPLE

```
unit SimpleMemTest;

interface

implementation

uses
  Windows;

var
  msg: string;

initialization
```

```
finalization
  if AllocMemCount > 0 then
  begin
    Str (AllocMemCount, msg);
    msg := msg + ' heap blocks left';
    MessageBox (0, PChar(msg), 'Memory Leak', MB_OK);
  end;
end.
```

TIP *When writing code that involves checking, implementing, or extending memory manager code, you have to avoid using any high-level functions, as they might affect the memory manager. For example, in the SimpleMemTest unit, I couldn't include the SysUtils unit because it allocates memory. I had to resort to the traditional Turbo Pascal* `Str` *function instead of Delphi's standard* `IntToStr` *conversion.*

This program is handy, but doesn't really help you understand what went wrong. For this purpose, powerful third-party tools are available (some of which have free trial versions), or you can refer to my custom memory manager, which tracks memory allocations (described in Appendix A of this book).

What's Next?

After the detailed description of forms and secondary forms in the previous chapters, I have focused on the architecture of applications, discussing both how Delphi's `Application` object works and how you can structure applications with multiple forms.

In particular, I've discussed MDI, visual form inheritance, and frames. Toward the end of the chapter I also discussed custom architectures, with form inheritance and interfaces. Now we can move forward to another key element of non-trivial Delphi applications: building custom components to use in your programs. I could write a book about this topic, so the description won't be exhaustive; but I'll offer a comprehensive overview.

Another element associated with the architecture of Delphi applications is the use of packages, which I'll introduce as a technology related to components but which really goes further. You can structure the code of a large application in multiple packages containing forms and other units. The development of programs based on multiple executable files, libraries, and packages, is discussed in Chapter 10.

After this step, I will begin delving into Delphi database programming, another key element of the Borland development environment and for many developers, the prime focus.

Chapter 9

Writing Delphi Components

Most Delphi programmers are probably familiar with using existing components, but at times it can also be useful to write your own components or to customize existing ones. One of the most interesting aspects of Delphi is that creating components is not much more difficult than writing programs. For this reason, even though this book is intended for Delphi application programmers and not Delphi tool writers, this chapter will discuss creating components and introduce Delphi add-ins, such as component and property editors.

This chapter gives you an overview of writing Delphi components and presents some examples. There is not enough space to present very complex components, but the ideas I've included cover all the basics and will get you started.

NOTE *You'll find more information about writing components in Chapter 17, "Writing Database Components," including how to build data-aware components.*

This chapter covers the following topics:

- Extending the Delphi library
- Writing packages
- Compound components
- Using interface-type properties
- Defining custom events
- Customizing existing components
- Using array properties
- Placing a dialog box in a component
- Collection properties
- Writing property and component editors

Extending the Delphi Library

Delphi components are classes, and the Visual Components Library (VCL) is the collection of all the classes defining Delphi components. You extend the VCL by writing new component classes in a package and installing it in Delphi. These new classes will be derived from one of the existing component-related classes or the generic `TComponent` class, adding new capabilities to those they inherit.

You can derive a new component from an existing component or from an *abstract component class*—one that does not correspond to a usable component. The VCL hierarchy includes many of these intermediate classes (often indicated with the `TCustom` prefix in their name) to let you choose a default behavior for your new component and change its properties.

Component Packages

Components are added to component packages. Each component package is basically a DLL (a dynamic link library) with a BPL extension (which stands for Borland Package Library).

Packages come in two flavors: design-time packages used by the Delphi IDE and run-time packages optionally used by applications. The design-only or run-only package option determines the package's type. When you attempt to install a package, the IDE checks whether it has the design-only or run-only flag, and decides whether to let the user install the package and whether it should be added to the list of run-time packages. Because there are two nonexclusive options, each with two possible states, there are four different kinds of component packages—two main variations and two special cases:

- Design-only component packages can be installed in the Delphi environment. These packages usually contain the design-time parts of a component, such as its property editors and the registration code. Often they also contain the components themselves, although this is not the most professional approach. The code of a design-only package's components is usually statically linked into the executable file, using the code of the corresponding Delphi Compiled Unit (DCU) files. Keep in mind, however, that it is also technically possible to use a design-only package as a run-time package.
- Run-only component packages are used by Delphi applications at run time. They cannot be installed in the Delphi environment, but they are automatically added to the list of run-time packages when they are required by a design-only package you install. Run-only packages usually contain the code of the component classes, but no design-time support (this is done to minimize the size of the component libraries you ship with your executable file). Run-only packages are important because they can be freely distributed with applications, but other Delphi programmers won't be able to install them in the environment to build new programs.
- Plain component packages (having neither the design-only nor the run-only option set) cannot be installed and will not be added to the list of run-time packages automatically. This option might make sense for utility packages used by other packages, but such packages are certainly rare.
- Packages with both flags set can be installed and are automatically added to the list of run-time packages. Usually these packages contain components requiring little or no design-time support (apart from the limited component registration code).

TIP *The filenames of Delphi's own design-only packages begin with the letters* DCL *(for example,* `DCLSTD60.BPL`*); filenames of run-only packages begin with the letters* VCL *(for example,* `VCL60.BPL`*). You can follow the same approach for your own packages, if you wish.*

In Chapter 1, "Delphi 7 and Its IDE," we discussed the effect of packages on the size of a program's executable file. Now we'll focus on building packages, because this is a required step in creating or installing components in Delphi.

When you compile a run-time package, you produce both a DLL with the compiled code (the BPL file) and a file with only symbol information (a DCP file), including no compiled machine code. The Delphi compiler uses the latter file to gather symbol information about the units that are part of the package without having access to the unit (DCU) files, which contain both the symbol information and the compiled machine code. This process reduces compilation time and allows you to distribute just the packages without the precompiled unit files. The precompiled units are still required to statically link the components into an application. Distribution of precompiled DCU files (or source code) may make sense depending on the kind of components you develop. You'll see how to create a package after we've discussed some general guidelines and built a component.

NOTE *DLLs are executable files containing collections of functions and classes, which can be used by an application or another DLL at run time. The typical advantage is that if many applications use the same DLL, only one copy needs to be on the disk or loaded in memory, and the size of each executable file will be much smaller. This is what happens with Delphi packages, as well. Chapter 10, "Libraries and Packages," looks at DLLs and packages in more detail.*

Rules for Writing Components

Some general rules govern the writing of components. You can find a detailed description of most of them in the *Delphi Component Writer's Guide* Help file, which is required reading for Delphi component writers.

Here is my own summary of the rules for component writers:

- Study the Delphi language with care. Particularly important concepts are inheritance, method overriding and overloading, the difference between public and published sections of a class, and the definition of properties and events. If you don't feel confident with the Delphi language or basic VCL concepts, you can refer to the overall description of the language and library presented in Part I of the book, particularly Chapters 2 ("The Delphi Programming Language") and 4 ("Core Library Classes").
- Study the structure of the VCL class hierarchy and keep a graph of the classes at hand (such as the one included with Delphi).
- Follow the standard Delphi naming conventions. There are several for components, as you will see, and following these rules makes it easier for other programmers to interact with your components and further extend them.
- Keep components simple, mimic other components, and avoid dependencies. These three rules basically mean that a programmer using your components should be able to use them as easily as preinstalled Delphi components. Use similar property, method, and event names whenever possible. If users don't need to learn complex rules about the use of your component (that is, if the dependencies between methods or properties are limited) and can access properties with meaningful names, they'll be happy.
- Use exceptions. When something goes wrong, the component should raise an exception. When you are allocating resources of any kind, you must protect them with `try/finally` blocks or destructor calls as appropriate.

- To complete a component, add a bitmap to it, which will be used by Delphi's Component Palette. If you intend for your component to be used by more than a few people, consider adding a Help file as well.
- Be ready to write *real* code and forget about the visual aspects of Delphi. Writing components generally means writing code without visual support (although Class Completion can speed up the coding of plain classes). The exception to this rule is that you can use frames to write components visually.

NOTE *You can also use a third-party component-writing tool to build your component or to speed up its development. The most powerful third-party tool I know of for creating Delphi components is the Class Developer Kit (CDK) from Eagle Software (*`www.eagle-software.com`*), but many others are available.*

The Base Component Classes

To build a new component, you generally begin with an existing one or with one of the VCL base classes. In either case, your component is in one of three broad component categories (introduced in Chapter 4), set by the three basic classes of the component hierarchy:

- `TWinControl` is the parent class of any component based on a window. Components that descend from this class can receive the input focus and get Windows messages from the system. You can also use their window handle when calling API functions. When creating a brand-new window control, you'll generally inherit from the derived class `TCustomControl`, which has a few extra useful features (particularly some support for painting the control).
- `TGraphicControl` is the parent class of visible components that have no Windows handle (which saves some Windows resources). These components cannot receive the input focus or respond to Windows messages directly. When creating a brand-new graphical control, you'll inherit directly from this class (which has a set of features very similar to `TCustomControl`).
- `TComponent` is the parent class of all components (including the controls) and can be used as a direct parent class for nonvisual components.

In the rest of the chapter, you will build some components using various parent classes, and we'll look at the differences among them. Let's begin with components inheriting from existing components or classes at a low level of the hierarchy, and then look at examples of classes inheriting directly from the ancestor classes just mentioned.

Building Your First Component

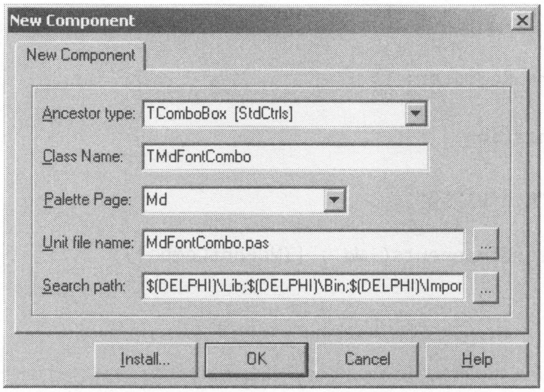
Building components is an important activity for Delphi programmers. Basically, any time you need the same behavior in two different places in an application, or in two different applications, you can place the shared code in a class—or, even better, in a component.

In this section, I'll introduce a couple of components to give you an idea of the steps required to build one. I'll also show you different things you can do to customize an existing component with a limited amount of code.

The Fonts Combo Box

Many applications have a toolbar with a combo box you can use to select a font. If you often use such a customized combo box, why not turn it into a component? Doing so will probably take less than a minute.

To begin, close any active projects in the Delphi environment and start the Component Wizard, either by choosing Component ➢ New Component, or by selecting File ➢ New ➢ Other to open the Object Repository and then choosing the component in the New page. As you can see, the Component Wizard requires the following information:



Ancestor type: TComboBox [StdCtrls]
Class Name: TMdFontCombo
Palette Page: Md
Unit file name: MdFontCombo.pas
Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Impor

- The name of the ancestor type: the component class you want to inherit from. In this case, use `TComboBox`.
- The name of the class of the new component you are building; use `TMdFontCombo`.
- The Component Palette page where you want to display the new component, which can be a new or an existing page. Create a new page, called *Md*.
- The filename of the unit where Delphi will place the source code of the new component; type **`MdFontCombo`**.
- The current search path (which should be set up automatically).

Click the OK button, and the Component Wizard will generate the source file shown in Listing 9.1 with the structure of your component. The Install button can be used to install the component in a package immediately. Let's look at the code first and then discuss the installation.

Listing 9.1: Code of the TMdFontCombo Class, Generated by the Component Wizard

```
unit MdFontCombo;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
type
  TMdFontCombo = class (TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md', [TMdFontCombo]);
end;

end.
```

One of the key elements of this listing is the class definition, which begins by indicating the parent class. The only other relevant portion is the `Register` procedure. As you can see, the Component Wizard does very little work.

WARNING *The* `Register` *procedure* must *be written with an uppercase* R. *This requirement is imposed for C++Builder compatibility (identifiers in C++ are case-sensitive).*

TIP *Use a naming convention when building components. All the components installed in Delphi should have different class names. For this reason, most Delphi component developers have chosen to add a two- or three-letter signature prefix to the names of their components. I've done the same, using* Md *(for* Mastering Delphi*) to identify components built in this book. The advantage of this approach is that you can install my* `TMdFontCombo` *component even if you've already installed a component named* `TFontCombo`*. Notice that the unit names must also be unique for all the components installed in the system, so I've applied the same prefix to the unit names.*

That's all it takes to build a component. Of course, this example doesn't include much code. You need only copy all the system fonts to the `Items` property of the combo box at startup. To do so, you might try to override the `Create` method in the class declaration, adding the statement `Items := Screen.Fonts`. However, this is not the correct approach. The problem is, you cannot access the combo box's `Items` property before the window handle of the component is available; the component cannot have a window handle until its `Parent` property is set; and that property isn't set in the constructor, but later.

For this reason, instead of assigning the new strings in the `Create` constructor, you must perform this operation in the `CreateWnd` procedure, which is called to create the window control after the component is constructed, its `Parent` property is set, and its window handle is available. Again, you execute the default behavior, and then you can write your custom code. I could have skipped the `Create` constructor and written all the code in `CreateWnd`, but I decided to use both startup methods to demonstrate the difference between them. Here is the declaration of the component class:

```
type
  TMdFontCombo = class (TComboBox)
  private
    FChangeFormFont: Boolean;
    procedure SetChangeFormFont(const Value: Boolean);
  public
    constructor Create (AOwner: TComponent); override;
    procedure CreateWnd; override;
    procedure Change; override;
  published
    property Style default csDropDownList;
    property Items stored False;
    property ChangeFormFont: Boolean
      read FChangeFormFont write SetChangeFormFont default True;
  end;
```

And here is the source code of its two methods executed at startup:

```
constructor TMdFontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
  FChangeFormFont := True;
end;

procedure TMdFontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items.Assign (Screen.Fonts);

  // grab the default font of the owner form
  if FChangeFormFont and Assigned (Owner) and (Owner is TForm)
   then
    ItemIndex := Items.IndexOf ((Owner as TForm).Font.Name);
end;
```

Notice that besides giving a new value to the component's `Style` property in the `Create` method, you redefine this property by setting a value with the `default` keyword. You have to do both operations because adding the `default` keyword to a property declaration has no direct effect on the property's initial value. Why specify a property's default value then? Because properties that have a value equal to the default are not streamed with the form definition (and they don't appear in the textual description of the form, the DFM file). The `default` keyword tells the streaming code that the component initialization code will set the value of that property.

TIP *It is important to specify a default value for a published property to reduce the size of the DFM files and, ultimately, the size of the executable files (which include the DFM files).*

The other redefined property, `Items`, is set as a property that should not be saved to the DFM file at all, regardless of the actual value. This behavior is obtained with the `stored` directive followed by the value False. The component and its window will be created again when the program starts, so it doesn't make sense to save in the DFM file information that will be discarded later (to be replaced with the new list of fonts).

NOTE *You can write the code of the* `CreateWnd` *method to copy the fonts to the combo box items only at run time, using statements such as* `if not (csDesigning in ComponentState)`. *But for this first component you are building, the less efficient but more straightforward method I've described offers a clearer illustration of the basic procedure.*

The third property, `ChangeFormFont`, is not inherited but introduced by the component. It is used to determine whether the current font selection in the combo box should specify the font of the form hosting the component. Again, this property is declared with a default value, which is set in the constructor. The `ChangeFormFont` property is used in the code of the `CreateWnd` method, shown earlier, to set up the initial selection of the combo depending on the font of the form hosting the component. This is generally the component's `Owner`, although I could also have walked the `Parent` tree looking for a form component. This code isn't perfect, but the `Assigned` and `is` tests provide some extra safety.

The `ChangeFormFont` property and the same `if` test play a key role in the `Changed` method, which in the base class triggers the `OnChange` event. By overriding this method, you provide a default behavior (which can be disabled by toggling the value of the property) but also allow the execution of the `OnChange` event, so that users of this class can fully customize its behavior. The final method, `SetChangeFormFont`, has been modified to refresh the form's font in case the property is being turned on. The complete code is as follows:

```
procedure TMdFontCombo.Change;
begin
  // assign the font to the owner form
  if FChangeFormFont and Assigned (Owner)
      and (Owner is TForm)
   then
    TForm (Owner).Font.Name := Text;
  inherited;
end;

procedure TMdFontCombo.SetChangeFormFont(const Value: Boolean);
begin
  FChangeFormFont := Value;
  // refresh font
  if FChangeFormFont then
    Change;
end;
```

Creating a Package

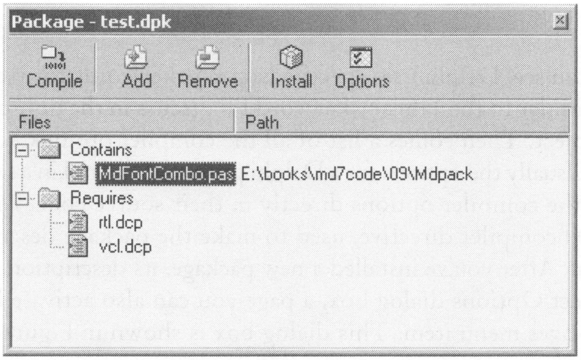
Now you have to install the component in the environment, using a package. For this example, you can either create a new package or use an existing one, such as the default user's package.

In either case, choose the Component ➢ Install Component menu command. The resulting dialog box has a page that lets you install the component into an existing package, and a page where you can create a new package. In the latter case, type in a filename and a description for the package. Clicking OK opens the Package Editor (see Figure 9.1), which has two parts:

- The Contains list indicates the components included in the package (or, to be more precise, the units defining those components).
- The Requires list indicates the packages required by this package. Your package will generally require the rtl and vcl packages (the main run-time library package and core VCL package), but it might also need the vcldb package (which includes most of the database-related classes) if the components of the new package do any database-related operations.

FIGURE 9.1

The Package Editor



NOTE *Since Delphi 6, package names aren't version specific, even if the compiled packages have a version number in the filename. See the section "Changing Project and Library Names" in Chapter 10 for more details about how this is achieved technically.*

Add the component to the new package you've just defined, and then compile the package and install it (using the two corresponding toolbar buttons in the Package Editor); the new component will immediately appear in the Md page of the Component Palette. The component unit file's `Register` procedure told Delphi where to install the new component. By default, the bitmap used will be the same as that of the parent class, because you haven't provided a custom bitmap (you will do this in later examples). Notice also that if you move the mouse over the new component, Delphi displays as a hint the name of the class without the initial letter *T*.

WHAT'S BEHIND A PACKAGE?

The Package Editor generates the source code for the package project: a special kind of DLL built in Delphi. The package project is saved in a file with the DPK (for Delphi PacKage) extension, displayed if you press the F12 key in the package editor. A typical package project looks like this:

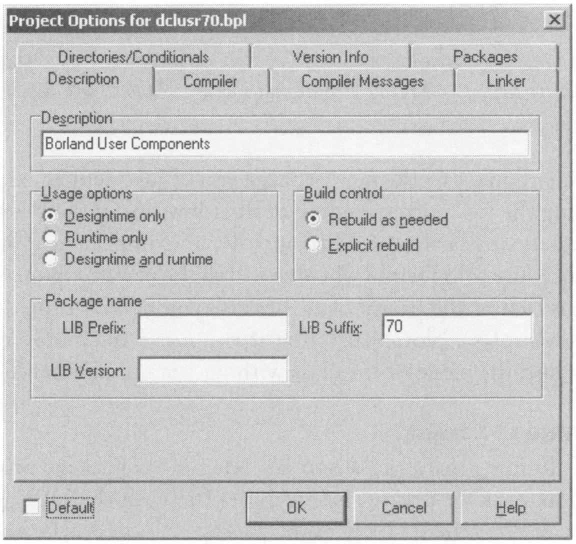
```
package MdPack;

{$R *.RES}
{$ALIGN ON}
{$BOOLEVAL OFF}
```

```
{$DEBUGINFO ON}
...
{$DESCRIPTION 'Mastering Delphi Package'}
{$IMPLICITBUILD ON}

requires
  vcl;

contains
  MdFontBox in 'MdFontBox.pas';

end.
```

As you can see, Delphi uses specific language keywords for packages. The first is the `package` keyword (similar to the `library` keyword I'll discuss in the next chapter), which introduces a new package project. Then comes a list of all the compiler options, some of which I've omitted from the listing. Usually the options for a Delphi project are stored in a separate file; packages, by contrast, include all the compiler options directly in their source code. Among the compiler options is a `DESCRIPTION` compiler directive, used to make the package description available to the Delphi environment. After you've installed a new package, its description will appear in the Packages page of the Project Options dialog box, a page you can also activate by selecting the Component ➢ Install Packages menu item. This dialog box is shown in Figure 9.2.

FIGURE 9.2

The Project Options for packages



Project Options for dclusr70.bpl

Description: Borland User Components

Usage options: ● Designtime only ○ Runtime only ○ Designtime and runtime

Build control: ● Rebuild as needed ○ Explicit rebuild

Package name: LIB Prefix: LIB Suffix: 70 LIB Version:

☐ Default

In addition to common directives like `DESCRIPTION`, there are other compiler directives specific to packages. The most common of these options are easily accessible through the Package Editor's Options button. After this list of options come the `requires` and `contains` keywords, which list the items displayed visually in the two pages of the Package Editor. Again, `requires` lists the packages required by the current package, and `contains` lists the units installed by this package.

Let's consider the technical effect of building a package. Besides the DPK file with the source code, Delphi generates a BPL file with the dynamic link version of the package and a DCP file with the symbol information. In practice, this DCP file is the sum of the symbol information for the DCU files of the units contained in the package.

At design time, Delphi requires both the BPL and DCP files, because the BPL file has the code of the components created on the design form and the symbol information required by the code insight technology. If you link the package dynamically (using it as a run-time package), the DCP file will also be used by the linker, and the BPL file should be shipped along with the application's main executable file. If you instead link the package statically, the linker refers to the DCU files, and you'll need to distribute only the final executable file.

For this reason, as a component designer, you should generally distribute at least the BPL file, the DCP file, and the DCU files of the units contained in the package and any corresponding DFM files, plus a Help file. As an option, of course, you can also make available the source code files of the package units (the PAS files) and of the package itself (the DPK file).

WARNING *By default, Delphi places all the compiled package files (BPL and DCP) not in the package source code's folder but under the* `\Projects\BPL` *folder. It does this so the IDE can easily locate the files, and the location creates no particular problem. However, when you have to compile a project using components declared in those packages, Delphi may complain that it cannot find the corresponding DCU files, which are stored in the package source code folder. You can solve this problem by indicating the package source code folder in the library path (specified in the Environment Options, which affect all projects) or by indicating it in the search path for the current project (in the Project Options). If you choose the first approach, placing different components and packages in a single folder may result in a real time savings.*

INSTALLING THE COMPONENTS CREATED IN THIS CHAPTER

Having built your first package, you can now begin using the component you've added to it. Before you do so, however, I should mention that I've extended the MdPack package to include all the components you will build in this chapter, including different versions of the same component. I suggest you install this package. The best approach is to copy it into a directory of your path, so that it will be available both to the Delphi environment and to the programs you build with it. I've collected all the component source code files and the package definition in a single subdirectory, called MdPack. This allows the Delphi environment (or a specific project) to refer to only one directory when looking for the package's DCU files. As suggested in the earlier warning, I could have collected all the components presented in the book in a single folder on the website; however, I decided that keeping the chapter-based organization was more understandable for readers.

Remember, if you compile an application using the packages as run-time libraries, you'll need to install these new libraries on your clients' computers. If you instead compile the programs by statically linking to the units contained in the package, the package library will be required only by the development environment and not by the users of your applications.

Using the Font Combo Box

Let's create a new Delphi program to test the Font combo box. Move to the Component Palette, select the new component, and add it to a new form. A traditional-looking combo box will appear.

However, if you open the Items property editor, you'll see a list of the fonts installed on your computer. To build a simple example, I added a Memo component to the form with some text in it. By leaving the `ChangeFormFont` property on, you don't need to write any other, as you'll see in the example. As an alternative, I could have turned off the property and handled the component's `OnChange` event with code like this:

```
Memo1.Font.Name := MdFontCombo1.Text;
```

The aim of this program is only to test the behavior of the new component you have built. The component is still not very useful—you could have added a couple of lines of code to a form to obtain the same effect—but looking at a couple of components should help you get an idea of what is involved in component building.

The Component Palette Bitmaps

Before installing a component, you can take one further step: define a bitmap for the Component Palette. If you fail to do so, the Palette uses the parent class's bitmap, or a default object's bitmap if the parent class is not an installed component. Defining a new bitmap for the component is easy, once you know the rules. You can create one with the Image Editor included in Delphi by starting a new project and selecting the Delphi Component Resource (DCR) project type.

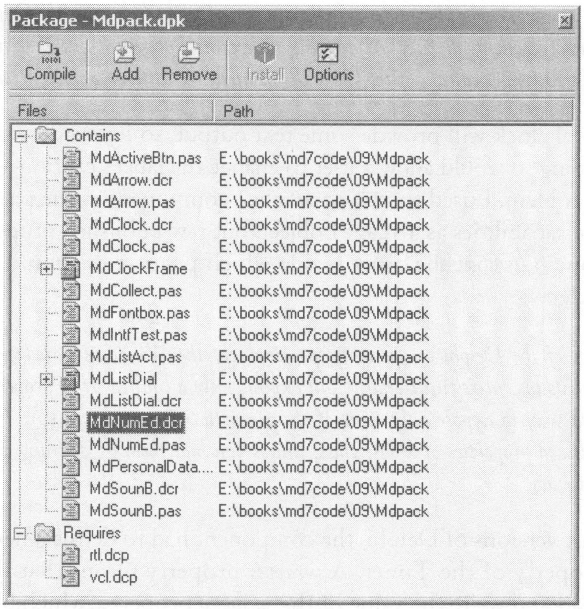
TIP *DCR files are standard RES files with a different extension. If you prefer, you can create them with any resource editor—including the Borland Resource Workshop, which is certainly a more powerful tool than the Delphi Image Editor. When you finish creating the resource file, rename the RES file to use a DCR extension.*

The resource file should have one (or more) bitmap, each 24×24 pixels. The only important rules refer to naming. In this case, the naming rules are not just a convention; they are required so that the IDE can find the image for a given component class:

- The name of the bitmap resource must match the name of the component, including the initial *T*. In this case, the name of the bitmap resource should be TMDFONTCOMBO. The name of the bitmap resource must be uppercase—this is mandatory.
- If you want the Package Editor to recognize and include the resource file, the name of the DCR file must match the name of the compiled unit that defines the component. In this case, the filename should be `MdFontBox.dcr`. If you manually include the resource file via a `$R` directive, you can give it any name you like, as well as use an RES extension and add multiple bitmaps in it.

When the bitmap for the component is ready, you can install the component in Delphi by using the Package Editor's Install Package toolbar button. After this operation, the Contains section of the editor will list both the component's PAS file and the corresponding DCR file. In Figure 9.3, you can see all the files (including the DCR files) of the final version of the MdPack package. If the DCR installation doesn't work properly, you can manually add the `{$R unitname.dcr}` statement in the package source code.

FIGURE 9.3 The Contains section of the Package Editor shows both the units that are included in the package and the component resource files.



Files	Path
Contains	
MdActiveBtn.pas	E:\books\md7code\09\Mdpack
MdArrow.dcr	E:\books\md7code\09\Mdpack
MdArrow.pas	E:\books\md7code\09\Mdpack
MdClock.dcr	E:\books\md7code\09\Mdpack
MdClock.pas	E:\books\md7code\09\Mdpack
MdClockFrame	E:\books\md7code\09\Mdpack
MdCollect.pas	E:\books\md7code\09\Mdpack
MdFontbox.pas	E:\books\md7code\09\Mdpack
MdIntfTest.pas	E:\books\md7code\09\Mdpack
MdListAct.pas	E:\books\md7code\09\Mdpack
MdListDial	E:\books\md7code\09\Mdpack
MdListDial.dcr	E:\books\md7code\09\Mdpack
MdNumEd.dcr	E:\books\md7code\09\Mdpack
MdNumEd.pas	E:\books\md7code\09\Mdpack
MdPersonalData....	E:\books\md7code\09\Mdpack
MdSounB.dcr	E:\books\md7code\09\Mdpack
MdSounB.pas	E:\books\md7code\09\Mdpack
Requires	
rtl.dcp	
vcl.dcp	

Creating Compound Components

Components don't exist in isolation. Programmers often use components in conjunction with other components, coding the relationship in one or more event handlers. An alternative approach is to write compound components, which can encapsulate this relationship and make it easy to handle. There are two different types of compound components:

Internal Components Created and managed by the main component, which may surface some of their properties and events.

External Components Connected using properties. Such a compound component automates the interaction of two separate components, which can be on the same or a different form or designer.

In both cases, development follows some standard rules.

A third, less-explored alternative, involves the development of component containers, which interact with the child controls. This is a more advanced topic, and I won't explore it here.

Internal Components

The next component I will focus on is a digital clock. This example has some interesting features. First, it embeds a component (a Timer) in another component; second, it shows the live-data approach: you'll be able to see a dynamic behavior (the clock time being updated) even at design time, as it happens, for example, with data-aware components.

NOTE *The first feature has become more relevant since Delphi 6, because the Object Inspector now allows you to expose properties of subcomponents directly. As a result, the example presented in this section has been modified (and simplified) compared to the Delphi 5 edition of the book. I'll mention the differences, when relevant.*

The digital clock will provide some text output, so I considered inheriting from the `TLabel` class. However, doing so would allow a user to change the label's caption—that is, the text of the clock. To avoid this problem, I used the `TCustomLabel` component as the parent class. A `TCustomLabel` object has the same capabilities as a `TLabel` object, but few published properties. In other words, a class that inherits from `TCustomLabel` can decide which properties should be available and which should remain hidden.

NOTE *Most of the Delphi components, particularly the Windows-based ones, have a* `TCustomXxx` *base class, which implements the entire functionality but exposes only a limited set of properties. Inheriting from these base classes is the standard way to expose only some of the properties of a component in a customized version. You cannot hide public or published properties of a base class, unless you hide them by defining a new property with the same name in the descendant class.*

With past versions of Delphi, the component had to define a new property, `Active`, wrapping the `Enabled` property of the Timer. A *wrapper* property means that the get and set methods of this property read and write the value of the *wrapped* property, which belongs to an internal component (a wrapper property generally has no local data). In this specific case, the code looks like this:

```
function TMdClock.GetActive: Boolean;
begin
  Result := FTimer.Enabled;
end;

procedure TMdClock.SetActive (Value: Boolean);
begin
  FTimer.Enabled := Value;
end;
```

Publishing Subcomponents

Beginning with Delphi 6, you can expose the entire subcomponent (the timer) in a property of its own that will be regularly expanded by the Object Inspector, allowing a user to set each of its subproperties and even to handle its events.

Here is the full type declaration for the `TMdClock` component, with the subcomponent declared in the private data and exposed as a published property (in the last line):

```
type
  TMdClock = class (TCustomLabel)
  private
    FTimer: TTimer;
  protected
    procedure UpdateClock (Sender: TObject);
```

```
public
  constructor Create (AOwner: TComponent); override;
published
  property Align;
  property Alignment;
  property Color;
  property Font;
  property ParentColor;
  property ParentFont;
  property ParentShowHint;
  property PopupMenu;
  property ShowHint;
  property Transparent;
  property Visible;
  property Timer: TTimer read FTimer;
end;
```

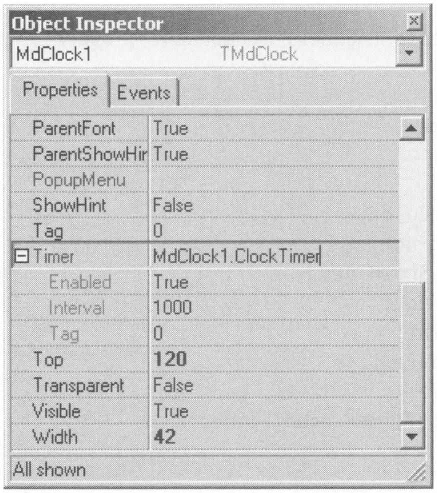
The `Timer` property is read-only, because I don't want users to select another value for this component in the Object Inspector (or detach the component by clearing the value of this property). Developing sets of subcomponents that can be used alternately is certainly possible, but adding write support for this property in a safe way is far from trivial (considering that the users of your component might not be expert Delphi programmers). So, I suggest you stick with read-only properties for subcomponents.

To create the Timer, you must override the clock component's constructor. The `Create` method calls the corresponding method of the base class and creates the Timer object, installing a handler for its `OnTimer` event:

```
constructor TMdClock.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  // create the internal timer object
  FTimer := TTimer.Create (Self);

  FTimer.Name := 'ClockTimer';
  FTimer.OnTimer := UpdateClock;
  FTimer.Enabled := True;
  FTimer.SetSubComponent (True);
end;
```

The code gives the component a name for display in the Object Inspector (see Figure 9.4) and calls the specific `SetSubComponent` method. You don't need a destructor; the `FTimer` object has the `TMDClock` component as owner (as indicated by the parameter of its `Create` constructor), so it will be destroyed automatically when the clock component is destroyed.

NOTE *In the previous code, the call to the* `SetSubComponent` *method sets an internal flag that's saved in the* `ComponentStyle` *property. The flag (csSubComponent) affects the streaming system, allowing the subcomponent and its properties to be saved in the DFM file. The streaming system by default ignores components that are not owned by the form.*



ParentFont	True
ParentShowHir	True
PopupMenu	
ShowHint	False
Tag	0
⊟ Timer	MdClock1.ClockTimer
Enabled	True
Interval	1000
Tag	0
Top	120
Transparent	False
Visible	True
Width	42

FIGURE 9.4 The Object Inspector can automatically expand subcomponents, showing their properties, as in the case of the `Timer` property of the TMdClock component.

The key piece of the component's code is the `UpdateClock` procedure, which is just one statement:

```
procedure TMdLabelClock.UpdateClock (Sender: TObject);
begin
  // set the current time as caption
  Caption := TimeToStr (Time);
end;
```

This method uses `Caption`, which is an unpublished property, so that a user of the component cannot modify it in the Object Inspector. This statement displays the current time continuously, because the method is connected to the Timer's `OnTimer` event.

NOTE *Events of subcomponents can be edited in the Object Inspector, so that a user can handle them. If you handle the event internally, as I did in* `TMdLabelClock`*, a user can override the behavior by handling the event, in this case* `OnTimer`*. In general, the solution is to define a derived class for the internal component, overriding its virtual methods, such as the* `TTimer` *class's* `Timer` *method. In this case, though, this technique won't work, because Delphi activates the timer only if an event handler is attached to it. If you override the virtual method and do not provide the event handler (as would be correct in a subcomponent), the timer won't work.*

External Components

When a component refers to an *external* component, it doesn't create this component itself (which is why this is called *external*). It is the programmer using the components that creates both of them separately (for example dragging them to a form from the Components Palette) and connects the two components using one of their properties. So we can say that a property of a component refers to an externally linked component. This property must be of a class type that inherits from `TComponent`.

To demonstrate, I've built a nonvisual component that can display data about a person on a label and refresh the data automatically. The component has these published properties:

```
type
  TMdPersonalData = class(TComponent)
    ...
  published
    property FirstName: string read FFirstName write SetFirstName;
    property LastName: string read FLastName write SetLastName;
    property Age: Integer read FAge write SetAge;
    property Description: string read GetDescription;
    property OutLabel: TLabel read FLabel write SetLabel;
  end;
```

There is some basic data plus a read-only `Description` property that returns all the information at once. The `OutLabel` property is connected with a local private field called `FLabel`. In the component's code, I've used this external label by means of the internal `FLabel` reference, as in the following:

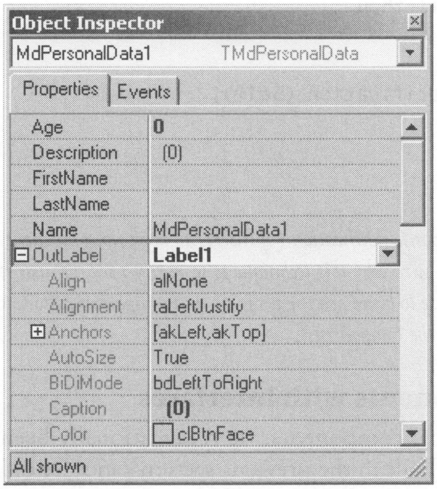
```
procedure TMdPersonalData.UpdateLabel;
begin
  if Assigned (FLabel) then
    FLabel.Caption := Description;
end;
```

This `UdpateLabel` method is triggered every time one of the other properties changes (as you can see at design time in Figure 9.5), as shown here:

```
procedure TMdPersonalData.SetFirstName(const Value: string);
begin
  if FFirstName <> Value then
  begin
    FFirstName := Value;
    UpdateLabel;
  end;
end;
```

FIGURE 9.5

A component referencing an external label at design time



MdPersonalData1 TMdPersonalData

Age	0
Description	(0)
FirstName	
LastName	
Name	MdPersonalData1
⊟OutLabel	Label1
Align	alNone
Alignment	taLeftJustify
⊞Anchors	[akLeft,akTop]
AutoSize	True
BiDiMode	bdLeftToRight
Caption	(0)
Color	☐ clBtnFace

All shown

Of course, you cannot use the label if it is not assigned; hence the need for the initial test. However, this test doesn't guarantee the label won't be used after it is destroyed (either at run time or at design time). When you write a component with a reference to an external component you need to override the `Notification` method in the component you are developing (the one with the external reference). This method is fired when a sibling component (one having the same owner) is created or destroyed. Consider the case of the `TMdPersonalData` class that receives the notification of the destruction (opRemove) of the Label component:

```
procedure TMdPersonalData.Notification(
  AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (AComponent = FLabel) and (Operation = opRemove) then
    FLabel := nil;
end;
```

This code is enough to avoid problems with components in the same form or designer (such as a data module), because when a component is destroyed, its owner notifies all the other components it owns (the siblings of the one being destroyed). To account for components connected across forms or data modules, however, you need to perform an extra step. Every component has an internal notification list of one or more components it must notify about its destruction. Your component can add itself to the notification list of components hooked to it (in this case, the label) by calling its `FreeNotification` method. So, even if the externally referenced label is on a different form, it will tell the component it is being destroyed by firing the `Notification` method (which is already handled and doesn't need to be updated):

```
procedure TMdPersonalData.SetLabel(const Value: TLabel);
begin
  if FLabel <> Value then
  begin
    FLabel := Value;
    if FLabel <> nil then
    begin
      UpdateLabel;
      FLabel.FreeNotification (Self);
    end;
  end;
end;
```

TIP You can also use the opposite notification (opInsert) to hook up components automatically as they are added to the same form or designer. I'm not sure why this technique is so rarely used, because it's helpful in many common situations. It is true that it makes more sense to build specific property and component editors to support design-time operations, rather than embed special code within the components.

Referring to Components with Interfaces

When referring to external components, we've traditionally been limited to a subhierarchy. For example, the component built in the previous section can refer only to objects of class `TLabel` or classes inheriting from `TLabel`, although it would make sense to also be able to output the data to other components. Delphi 6 added support for an interesting feature that has the potential to revolutionize some areas of the VCL: interface-type component references.

NOTE *This feature is used sparingly in Delphi 7. As it is probably too late to update Delphi's data-aware components architecture using interfaces, all we can hope is that this will be used to express other future complex relationships within the library.*

If you have components supporting a given interface (even if they are not part of the same subhierarchy), you can declare a property with an interface type and assign any of those components to it. For example, suppose you have a nonvisual component attached to a control for its output, similar to what I did in the previous section. I used a traditional approach and hooked the component to a label, but you can now define an interface as follows:

```
type
  IMdViewer = interface
    ['{9766860D-8E4A-4254-9843-59B98FEE6C54}']
    procedure View (const str: string);
  end;
```

A component can use this *viewer* interface to provide output to another control (of any type). Listing 9.2 shows how to declare a component that uses this interface to refer to an external component.

LISTING 9.2: A COMPONENT THAT REFERS TO AN EXTERNAL COMPONENT USING AN INTERFACE

```
type
  TMdIntfTest = class(TComponent)
  private
    FViewer: IViewer;
    FText: string;
    procedure SetViewer(const Value: IViewer);
    procedure SetText(const Value: string);
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  published
    property Viewer: IViewer read FViewer write SetViewer;
    property Text: string read FText write SetText;
  end;

{ TMdIntfTest }

procedure TMdIntfTest.Notification(AComponent: TComponent;
  Operation: TOperation);
var
  intf: IMdViewer;
begin
  inherited;
  if (Operation = opRemove) and
    (Supports (AComponent, IMdViewer, intf)) and (intf = FViewer) then
  begin
    FViewer := nil;
  end;
end;

procedure TMdIntfTest.SetText(const Value: string);
begin
  FText := Value;
  if Assigned (FViewer) then
    FViewer.View(FText);
end;

procedure TMdIntfTest.SetViewer(const Value: IMdViewer);
var
  iComp: IInterfaceComponentReference;
begin
  if FViewer <> Value then
  begin
    FViewer := Value;
    FViewer.View(FText);
    if Supports (FViewer, IInterfaceComponentReference, iComp) then
      iComp.GetComponent.FreeNotification(Self);
  end;
end;
```

The use of an interface implies two relevant differences, compared to the traditional use of a class type to refer to an external component. First, in the `Notification` method, you must extract the interface from the component passed as a parameter and compare it to the interface you already hold. Second, to call the `FreeNotification` method, you must see whether the object passed as the parameter supports the `IInterfaceComponentReference` interface. This is declared in the `TComponent` class and provides a way to refer back to the component (`GetComponent`) and call its methods. Without this help you would have to add a similar method to your custom interface, because when you extract an interface from an object, there is no automatic way to refer back to the object.

Now that you have a component with an interface property, you can assign to it any component (from any portion of the VCL hierarchy) by adding the `IViewer` interface to it and implementing the `View` method. Here is an example:

```
type
  TViewerLabel = class (TLabel, IViewer)
  public
    procedure View(str: String);
  end;

procedure TViewerLabel.View(const str: String);
begin
  Caption := str;
end;
```

BUILDING COMPOUND COMPONENTS WITH FRAMES

Instead of building the compound component in code and hooking up the timer event manually, you can obtain a similar effect by using a frame. Frames make the development of compound components with custom event handlers a visual operation, and thus simpler. You can share this frame by adding it to the Repository or by creating a template using the Add to Palette command on the frame's shortcut menu.

As an alternative, you might want to share the frame by placing it in a package and registering it as a component. Technically, this technique is not difficult: You add a `Register` procedure to the frame's unit, add the unit to a package, and build it. The new component/frame appears in the Component Palette like any other component. When you place this component/frame on a form, you'll see its subcomponents. You cannot select these subcomponents with a mouse click in the Form Designer, but you can do so in the Object TreeView. However, any change you make to these components at design time will be lost when you run the program or save and reload the form, because the changes to those subcomponents aren't streamed, unlike what happens with standard frames you place inside a form).

If this is not what you expect, I've found a *reasonable* way to use frames in packages, demonstrated by the MdFramedClock component (part of the examples for this chapter on the Sybex website). The components owned by the frame are turned into subcomponents by calling the `SetSubComponent` method. I also exposed the internal components with properties, even though this step isn't compulsory (they can be selected in the Object TreeView). Here is the component's declaration and the code for its methods:

```
type
  TMdFramedClock = class(TFrame)
    Label1: TLabel;
    Timer1: TTimer;
    Bevel1: TBevel;
      procedure Timer1Timer(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
  published
    property SubLabel: TLabel read Label1;
    property SubTimer: TTimer read Timer1;
  end;constructor TMdFramedClock.Create(AOwner: TComponent);
begin
  inherited;
  Timer1.SetSubComponent (True);
  Label1.SetSubComponent (True);
end;procedure TMdFramedClock.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := TimeToStr (Time);
end;
```

Continued on next page

BUILDING COMPOUND COMPONENTS WITH FRAMES (*continued*)

In contrast to the clock component built earlier, there is no need to set up the properties of the timer, or to connect the timer event to its handler function manually, because this is done visually and saved in the DFM file of the frame. Notice also that I haven't exposed the Bevel component (I haven't called `SetSubComponent` on it). I did this on purpose so you can experiment with this fault behavior: try editing it at design time and see that all the changes are lost, as I mentioned earlier.

After you install this frame/component, you can use it in any application. In this case, as soon as you drop the frame on the form, the timer will begin to update the label with the current time. However, you can still handle its `OnTimer` event, and the Delphi IDE (recognizing that the component is in a frame) will create a method with this predefined code:

```
procedure TForm1.MdFramedClock1Timer1Timer(Sender: TObject);
begin
  MdFramedClock1.Timer1Timer(Sender);
end;
```

As soon as this timer is connected (even at design time) the live clock will stop, because its original event handler is disconnected. After you compile and run the program, however, the original behavior will be restored (at least, if you don't delete the previous line); your extra custom code will be executed as well. This behavior is exactly what you expect from frames. You can find a complete demo of the use of this frame/component in the FrameClock example.

This approach is still far from linear. It is much better than in past versions of Delphi, where frames inside packages were unusable, but it isn't worth the effort. If you work alone or with a small team, it's better to use plain frames stored in the Repository. In larger organizations or to distribute your frames to a larger audience, most people will prefer to build their components the traditional way, without trying to use frames. I hope that Borland will address more complete support for the visual development of packaged components based on frames.

A Complex Graphical Component

In this section, I'll demonstrate how to build a graphical Arrow component. You can use such a component to indicate a flow of information or an action. This component is quite complex, so I'll show you the various steps instead of looking directly at the complete source code. The component I've added to the MdPack package is the final version of this process, which demonstrates several important concepts:

- The definition of new enumerated properties, based on custom enumerated data types.
- The implementation of the component's `Paint` method, which provides its user interface and should be generic enough to accommodate all the possible values of the various properties, including its `Width` and `Height`. The `Paint` method plays a substantial role in this graphical component.
- The use of properties of `TPersistent`-derived classes, such as `TPen` and `TBrush`, as well as the issues related to their creation and destruction and to handling their `OnChange` events internally in your component.
- The definition of a custom event handler for the component, which responds to user input (in this case, a double-click on the point of the arrow). This requires direct handling of Windows messages and the use of the Windows API for graphic regions.

- The registration of properties in Object Inspector categories and the definition of a custom category.

Defining an Enumerated Property

After generating the new component with the Component Wizard and choosing `TGraphicControl` as the parent class, you can begin to customize the component. The arrow can point in any of four directions: up, down, left, or right. An enumerated type expresses these choices:

```
type
  TMdArrowDir = (adUp, adRight, adDown, adLeft);
```

This enumerated type defines a private data member of the component, a parameter of the procedure used to change it, and the type of the corresponding property.

The `ArrowHeight` property determines the size of the arrowhead, and the `Filled` property specifies whether to fill the arrowhead with color:

```
type
  TMdArrow = class (TGraphicControl)
  private
    FDirection: TMdArrowDir;
    FArrowHeight: Integer;
    FFilled: Boolean;
    procedure SetDirection (Value: TMd4ArrowDir);
    procedure SetArrowHeight (Value: Integer);
    procedure SetFilled (Value: Boolean);
  published
    property Width default 50;
    property Height default 20;
    property Direction: TMd4ArrowDir
      read FDirection write SetDirection default adRight;
    property ArrowHeight: Integer
      read FArrowHeight write SetArrowHeight default 10;
    property Filled: Boolean read FFilled write SetFilled default False;
```

NOTE A graphic control has no default size, so when you place it in a form, its size will be a single pixel. For this reason, it is important to add a default value for the `Width` and `Height` properties and set the class fields to the default property values in the class constructor.

The three custom properties are read directly from the corresponding field and are written using three `Set` methods, all having the same standard structure:

```
procedure TMdArrow.SetDirection (Value: TMdArrowDir);
begin
  if FDirection <> Value then
  begin
    FDirection := Value;
    ComputePoints;
```

```
      Invalidate;
    end;
  end;
```

Notice that you ask the system to repaint the component (by calling `Invalidate`) only if the property is really changing its value and after calling the `ComputePoints` method, which computes the triangle delimiting the arrowhead. Otherwise, the code is skipped and the method ends immediately. This code structure is common, and you will use it for most of the `Set` procedures of properties.

You must also remember to set the properties' default values in the component's constructor:

```
constructor TMdArrow.Create (AOwner: TComponent);
begin
  // call the parent constructor
  inherited Create (AOwner);
  // set the default values
  FDirection := adRight;
  Width := 50;
  Height := 20;
  FArrowHeight := 10;
  FFilled := False;
```

As mentioned before, the default value specified in the property declaration is used only to determine whether to save the property's value to disk. The `Create` constructor is defined in the public section of the new component's type definition, and the constructor is marked by the `override` keyword, as it replaces the virtual `Create` constructor of `TComponent`. It is fundamental to remember the `override` specifier; otherwise, when Delphi creates a new component of this class, it will call the base class's constructor, rather than the one you've written for your derived class.

PROPERTY-NAMING CONVENTIONS

In the definition of the Arrow component, notice the use of several naming conventions for properties, access methods, and fields. Here is a summary:

- A property should have a meaningful and readable name.
- When a private data field is used to hold the value of a property, the field should be named with an uppercase *F* (field) at the beginning, followed by the name of the corresponding property.
- When a procedure is used to change the value of the property, the function should have the word *Set* at the beginning, followed by the name of the corresponding property.
- A corresponding function used to read the property should have the word *Get* at the beginning, again followed by the property name.

These are just guidelines to make your programs more readable. The compiler doesn't enforce them. These conventions are described in the *Delphi Component Writers' Guide* and are followed by Delphi's class completion mechanism.

Writing the *Paint* Method

Drawing the arrow in the various directions and with the various styles requires a fair amount of code. To perform custom painting, you override the `Paint` method and use the protected `Canvas` property.

Instead of computing the position of the arrowhead points in drawing code that will be executed often, I've written a separate function to compute the arrowhead area and store it in an array of points defined among the private fields of the component:

```
FArrowPoints: array [0..3] of TPoint;
```

These points are determined by the `ComputePoints` private method, which is called every time a component property changes. Here is an excerpt of its code:

```
procedure TMdArrow.ComputePoints;
var
  XCenter, YCenter: Integer;
begin
  // compute the points of the arrowhead
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
  case FDirection of
    adUp: begin
      FArrowPoints [0] := Point (0, FArrowHeight);
      FArrowPoints [1] := Point (XCenter, 0);
      FArrowPoints [2] := Point (Width-1, FArrowHeight);
    end;
  // and so on for the other directions
```

The code computes the center of the component area (dividing the `Height` and `Width` properties by two) and then uses the center to determine the position of the arrowhead. In addition to changing the direction or other properties, you need to refresh the position of the arrowhead when the size of the component changes. You can override the `SetBounds` method of the component, which is called by VCL every time the `Left`, `Top`, `Width`, and `Height` properties of a component change:

```
procedure TMdArrow.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited SetBounds (ALeft, ATop, AWidth, AHeight);
  ComputePoints;
end;
```

Once the component knows the position of the arrowhead, its painting code becomes simpler. Here is an excerpt of the `Paint` method:

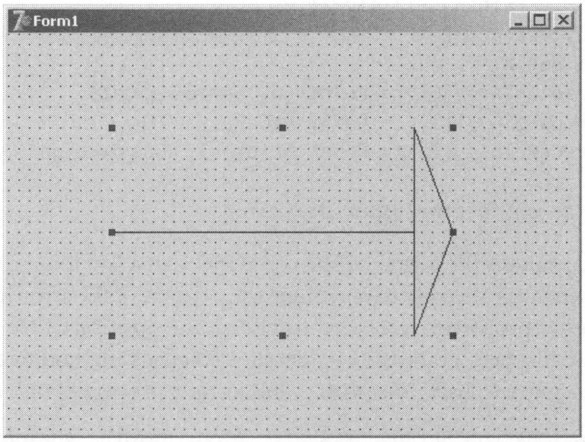
```
procedure TMdArrow.Paint;
var
  XCenter, YCenter: Integer;
begin
  // compute the center
  YCenter := (Height - 1) div 2;
  XCenter := (Width - 1) div 2;
```

```
    // draw the arrow line
    case FDirection of
      adUp: begin
        Canvas.MoveTo (XCenter, Height-1);
        Canvas.LineTo (XCenter, FArrowHeight);
      end;
      // and so on for the other directions
    end;

    // draw the arrow point, eventually filling it
    if FFilled then
      Canvas.Polygon (FArrowPoints)
    else
      Canvas.PolyLine (FArrowPoints);
  end;
```

You can see an example of the output of this component in Figure 9.6.

FIGURE 9.6

The output of the Arrow component



Adding *TPersistent* Properties

To make the output of the component more flexible, I've added to it two new properties, `Pen` and `Brush`, defined with a class type (a `TPersistent` data type, which defines objects that Delphi can automatically stream). These properties are a little more complex to handle, because the component now has to create and destroy these internal objects. This time, however, you also export the internal objects using properties, so that users can directly change these internal objects from the Object Inspector. To update the component when these subobjects change, you'll also need to handle their internal `OnChange` property. Here is the definition of the `Pen` property and the other changes to the definition of the component class (the code for the `Brush` property is similar):

```
type
  TMdArrow = class (TGraphicControl)
  private
```

```
    FPen: TPen;
    ...
    procedure SetPen (Value: TPen);
    procedure RepaintRequest (Sender: TObject);
  published
    property Pen: TPen read FPen write SetPen;
  end;
```

You first create the object in the constructor and set its `OnChange` event handler:

```
constructor TMdArrow.Create (AOwner: TComponent);
begin
  ...
  // create the pen and the brush
  FPen := TPen.Create;
  // set a handler for the OnChange event
  FPen.OnChange := RepaintRequest;
end;
```

These `OnChange` events are fired when one of the properties of the pen changes; all you have to do is to ask the system to repaint your component:

```
procedure TMdArrow.RepaintRequest (Sender: TObject);
begin
  Invalidate;
end;
```

You must also add a destructor to the component, to remove the graphical object from memory (and free its system resources). All the destructor has to do is call the Pen object's `Free` method.

A property related to persistent objects requires special handling: Instead of copying the pointer to the object, you have to copy the internal data of the object passed as a parameter. The standard `:=` operation copies the pointer, so in this case you have to use the `Assign` method:

```
procedureTMdArrow.SetPen (Value: TPen);
begin
  FPen.Assign(Value);
  Invalidate;
end;
```

Many `TPersistent` classes have an `Assign` method you should use when you need to update the data of these objects. Now, to use the pen for the drawing, you must modify the `Paint` method, setting the corresponding property of the component `Canvas` to the value of the internal object before drawing a line (see the example of the component's new output in Figure 9.7):

```
procedure TMdArrow.Paint;
begin
  // use the current pen
  Canvas.Pen := FPen;
```

As the `Canvas` uses a setter routine to `Assign` the pen object, you're not simply storing a reference to the pen in a field of the `Canvas`, but you are copying all of its data. This means that you can freely destroy the local Pen object (`FPen`) and that modifying `FPen` won't affect the canvas until `Paint` is called and the code above is executed again.

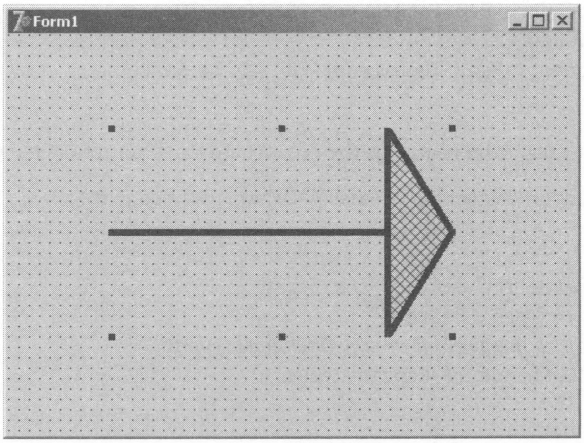


FIGURE 9.7 The output of the Arrow component with a thick pen and a special hatch brush

Defining a Custom Event

To complete the development of the Arrow component, let's add a custom event. Most of the time, new components use the events of their parent classes. For example, in this component, I've made some standard events available by redeclaring them in the published section of the class:

```
type
  TMdArrow = class (TGraphicControl)
  published
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
```

Thanks to this declaration, the events (originally declared in a parent class) will be available in the Object Inspector when the component is installed.

Sometimes, however, a component requires a custom event. To define a new event, you first need to ensure that there is already a method pointer type suitable for use by the event; if not, you need to define a new event type. This type is a method pointer type (see Chapter 5, "Visual Controls," for details). In both cases, you need to add to the class a field of the event's type: here is the definition I've added in the private section of the `TMdArrow` class:

```
FArrowDblClick: TNotifyEvent;
```

I've used the `TNotifyEvent` type, which has only a `Sender` parameter and is used by Delphi for many events, including `OnClick` and `OnDblClick` events. Using this field I've defined a published property, with direct access to the field:

```
property OnArrowDblClick: TNotifyEvent
  read FArrowDblClick write FArrowDblClick;
```

(Notice again the standard naming convention, with event names starting with *On.*) The `fArrowDblClick` method pointer is activated (executing the corresponding function) inside the specific `ArrowDblClick` dynamic method. This happens only if an event handler has been specified in the program that uses the component:

```
procedure TMdArrow.ArrowDblClick;
begin
  if Assigned (FArrowDblClick) then
    FArrowDblClick (Self);
end;
```

TIP *The use of* `Self` *as parameter of the invocation of the event handler method ensures that when the method is called its* `Sender` *parameter would actually refer to the object that fired the event, which you generally expect as a component user.*

USING LOW-LEVEL WINDOWS API CALLS

The `fArrowDblClick` method is defined in the protected section of the type definition to allow future descendant classes to both call and change it. Basically, this method is called by the handler of the `wm_LButtonDblClk` Windows message, but only if the double-click took place inside the arrow's point. To test this condition, you can use some of the Windows API's region functions.

NOTE *A region is a screen area enclosed by any shape. For example, you can build a polygonal region using the three vertices of the arrow-point triangle. The only problem is that to fill the surface properly, you must define an array of* `TPoints` *in a clockwise direction (see the description of the* `CreatePolygonalRgn` *in the Windows API Help for the details of this strange approach). That's what I did in the* `ComputePoints` *method.*

Once you have defined a region, you can use the `PtInRegion` API call to test whether the point where the double-click occurred is inside the region. The complete source code for this procedure is as follows:

```
procedure TMdArrow.WMLButtonDblClk (
  var Msg: TWMLButtonDblClk); // message wm_LButtonDblClk;
var
  HRegion: HRgn;
begin
  // perform default handling
  inherited;

  // compute the arrowhead region
  HRegion := CreatePolygonRgn (FArrowPoints, 3, WINDING);
  try // check whether the click took place in the region
    if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then
      ArrowDblClick;
  finally
    DeleteObject (HRegion);
  end;
end;
```

THE CLX VERSION: CALLING QT NATIVE FUNCTIONS

The previous code won't be portable to Linux and makes no sense for the CLX/Qt version of the component. If you want to build a similar component for the CLX class library, you can replace the Win32 API calls with direct (low-level) calls to the Qt layer, creating an object of the `QRegion` class, as in the following listing:

```
procedure TMdArrow.DblClick;
var
  HRegion: QRegionH;
  MousePoint: TPoint;
begin
  // perform default handling
  inherited;
  // compute the arrow head region
  HRegion := QRegion_create (PPointArray(FArrowPoints), True);
  try
    // get the current mouse position
    GetCursorPos (MousePoint);
    MousePoint := ScreenToClient(MousePoint);
    // check whether the click took place in the region
    if QRegion_contains(HRegion, PPoint(@MousePoint)) then
      ArrowDblClick;
  finally
    QRegion_destroy(HRegion);
  end;
end;
```

Registering Property Categories

You've added to this component some custom properties and a new event. If you arrange the properties in the Object Inspector by category, all the new elements will appear in the generic Miscellaneous category. Of course, this is far from ideal, but you can easily register the new properties in one of the available categories.

You can register a property (or an event) in a category by calling one of the four overloaded versions of the `RegisterPropertyInCategory` function, defined in the DesignIntf unit. When calling this function, you indicate the name of the category, and you can specify the property name, its type, or the property name and the component it belongs to. For example, you can add the following lines to the `Register` procedure of the unit to register the `OnArrowDblClick` event in the Input category and the `Filled` property in the Visual category:

```
uses
  DesignIntf;

procedure Register;
begin
  RegisterPropertyInCategory ('Input', TMdArrow, 'OnArrowDblClick');
  RegisterPropertyInCategory ('Visual', TMdArrow, 'Filled');
end;
```

The first parameter is a string indicating the category name—a much simpler solution than the original Delphi 5 approach of using category classes. You can define a new category in a straightforward manner by passing its name as the first parameter of the `RegisterPropertyInCategory` function:

```
RegisterPropertyInCategory ('Arrow', TMdArrow, 'Direction');
RegisterPropertyInCategory ('Arrow', TMdArrow, 'ArrowHeight');
```

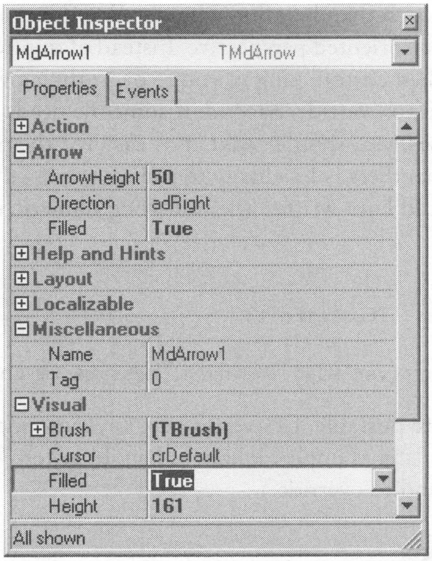
Creating a new category for the specific properties of your component can make it much simpler for a user to locate its specific features. Notice, though, that because you rely on the DesignIntf unit, you should compile the unit containing these registrations in a design-time package, not a run-time package (the required DesignIde unit cannot be distributed). For this reason, I've written this code in a separate unit from the one defining the component and added the new unit (MdArrReg) to the package MdDesPk, including all the design-time-only units; this approach is discussed later, in the section "Installing the Property Editor."

WARNING *It's debatable whether using a category for the specific properties of a component is a good idea. On one hand, a user of the component can easily spot specific properties. At the same time, some of the new properties may not pertain to any of the existing categories. On the other hand, categories can be overused. If every component introduces new categories, users may be confused. You also face the risk of having as many categories as there are properties.*

Notice that my code registers the `Filled` property in two different categories. This is not a problem, because the same property can show up multiple times in the Object Inspector under different groups, as you can see in Figure 9.8. To test the arrow component, I've written the ArrowDemo program, which allows you to modify most of its properties at run time. This type of test is important after you have written a component or while you are writing it.

FIGURE 9.8

The Arrow component defines a custom property category, Arrow, as you can see in the Object Inspector. Notice that properties can be visible in multiple section, such as the Filled property in this case.



MdArrow1 TMdArrow

⊞Action
⊟Arrow
ArrowHeight 50
Direction adRight
Filled True
⊞Help and Hints
⊞Layout
⊞Localizable
⊟Miscellaneous
Name MdArrow1
Tag 0
⊟Visual
⊞Brush (TBrush)
Cursor crDefault
Filled True
Height 161

NOTE *The* Localizable *property category has a special role related to the use of the ITE (Integrated Translation Environment). When a property is part of this category, its value is listed in the Translation Environment as a property that can be translated into another language.*

Customizing Windows Controls

One of the most common ways of customizing existing components is to add predefined behavior to their event handlers. Every time you need to attach the same event handler to components of different forms, you should consider adding the event code to a descendant class of the component. An obvious example is edit boxes that accept only numeric input. Instead of attaching a common `OnChar` event handler to each edit box, you can define a new component.

However, this component won't handle the event; events are for component users only. Instead, the component can either handle the Windows message directly or override a method, often called a *second-level* message handler. The former technique was commonly used in the past, but it makes a component specific to the Windows platform. To create a component that's portable to CLX and Linux—and, in the future, to the .NET architecture—you should avoid low-level Windows messages and instead override virtual methods of the base component and control classes.

NOTE *When most VCL components handle a Windows message, they call a* second-level *message handler (usually a* dynamic *method), instead of executing code directly in the message-response method. This approach makes it easier for you to customize the component in a derived class. Typically, a second-level handler will do its own work and then call any event handler the component user has assigned. So, you should always call* `inherited` *to let the component fire the event as expected.*

In addition to portability, there are other reasons why overriding existing second-level handlers is generally a better approach than handling straight Windows messages. First, this technique is more sound from an object-oriented perspective. Instead of duplicating the message-response code from the base class and then customizing it, you're overriding a virtual method call that the VCL designers planned for you to override. Second, if someone needs to derive another class from one of your component classes, you should make it as easy for them to customize as possible, and overriding second-level handlers is less likely to induce errors (if only because you're writing less code). For example, I could have written the following numeric edit box control by handling the `wm_Char` system message:

```
type
  TMdNumEdit = class (TCustomEdit)
  public
    procedure WmChar (var Msg: TWmChar); message wm_Char;
```

However, the code is more portable if I override the `KeyPress` method, as I've done in the code of the next component. (In a later example I'll have to handle custom Windows messages, because there is no corresponding method to override.)

The Numeric Edit Box

To customize an edit box component to restrict the input it will accept, all you need to do is override its `KeyPress` method, which is called when the component receives the `wm_Char` Windows message.

Here is the code for the `TMdNumEdit` class:

```
type
  TMdNumEdit = class (TCustomEdit)
  private
    FInputError: TNotifyEvent;
  protected
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    procedure KeyPress(var Key: Char); override;
  public
    constructor Create (Owner: TComponent); override;
  published
    property OnInputError: TNotifyEvent read FInputError write FInputError;
    property Value: Integer read GetValue write SetValue default 0;
    property AutoSelect;
    property AutoSize;
    // and so on...
```

This component inherits from `TCustomEdit` instead of `TEdit` so that it can hide the `Text` property and surface the Integer `Value` property instead. Notice that you don't create a new field to store this value, because you can use the existing (but now unpublished) `Text` property. To do so, you convert the numeric value to and from a text string. The `TCustomEdit` class (actually, the Windows control it wraps) automatically paints the information from the `Text` property on the surface of the component:

```
function TMdNumEdit.GetValue: Integer;
begin
  // set to 0 in case of error
  Result := StrToIntDef (Text, 0);
end;

procedure TMdNumEdit.SetValue (Value: Integer);
begin
  Text := IntToStr (Value);
end;
```

The most important method is the redefined `KeyPress` method, which filters out all the nonnumeric characters and fires a specific event in case of an error:

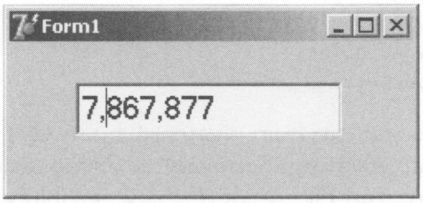
```
procedure TMdNumEdit.KeyPress (var Msg: TWmChar);
begin
  if not (Key in ['0'..'9']) and not (Key = #8) then
  begin
    Key := #0; // pretend that nothing was pressed
    if Assigned (FInputError) then
      FInputError (Self);
  end
  else
    inherited;
end;
```

This method checks each character as the user enters it, testing for numerals and the Backspace key (which has an ASCII value of 8). The user should be able to use Backspace in addition to the system keys (the arrow keys and Del), so you need to check for that value.

Now, place this component on a form, type something in the edit box, and see how it behaves. You might also want to attach a method to the `OnInputError` event to provide feedback to the user when an incorrect key is pressed.

A NUMERIC EDIT WITH THOUSANDS SEPARATORS

As a further extension of the example, when the user types large numbers (stored internally as floating point numbers, which compared to integers can be larger and have decimal digits) it would be nice for the thousands separators to automatically appear and update themselves as required by the input:



7,867,877

You can do this by overriding the internal `Change` method and formatting the number properly. There are only a couple of small problems to consider. The first is that to format the number you need to have a string containing a number, but the text in the edit box is not a numeric string Delphi recognizes, as it has thousands of separators and cannot be converted to a number directly. I've written a modified version of the `StringToFloat` function, called `StringToFloatSkipping`, to accomplish this conversion.

The second small problem is that if you modify the text in the edit box, the current position of the cursor will be lost. So, you need to save the original cursor position, reformat the number, and then reapply the cursor position—considering that if a separator has been added or removed, the cursor position should change accordingly.

All these considerations are summarized by the following complete code for the `TMdThousandEdit` class:

```
type
  TMdThousandEdit = class (TMdNumEdit)
  public
    procedure Change; override;
  end;

function StringToFloatSkipping (s: string): Extended;
var
  s1: string;
  I: Integer;
begin
  // remove non-numbers
  s1 := '';
```

```
    for i := 1 to length (s) do
     if s[i] in ['0'..'9'] then
       s1 := s1 + s[i];
    Result := StrToFloat (s1);
  end;

  procedure TMdThousandEdit.Change;
  var
    CursorPos, // original position of the cursor
    LengthDiff: Integer; // number of new separators (+ or -)
  begin
    if Assigned (Parent) then
    begin
      CursorPos := SelStart;
      LengthDiff := Length (Text);
      Text := FormatFloat ('#,###',
        StringToFloatSkipping (Text));
      LengthDiff := Length (Text) - LengthDiff;
      // move the cursor to the proper position
      SelStart := CursorPos + LengthDiff;
    end;
    inherited;
  end;
```

The Sound Button

The next component, `TMdSoundButton`, plays one sound when you press the button and another sound when you release it. The user specifies each sound by modifying two string properties that name the appropriate WAV files for the respective sounds. Once again, you need to intercept some system messages (`wm_LButtonDown` and `wm_LButtonUp`) or override the appropriate *second-level* handler.

Here is the code for the `TMdSoundButton` class, with the two protected methods and the two string properties that identify the sound files, mapped to private fields because you don't need to do anything special when the user changes those properties:

```
type
  TMdSoundButton = class(TButton)
  private
    FSoundUp, FSoundDown: string;
  protected
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  published
    property SoundUp: string read FSoundUp write FSoundUp;
    property SoundDown: string read FSoundDown write FSoundDown;
  end;
```

Here is the code for one of the two methods:

```
uses
  MMSystem;

procedure TMdSoundButton.MouseDown(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
  PlaySound (PChar (FSoundDown), 0, snd_Async);
end;
```

Notice that you call the inherited version of the methods *before* you do anything else. For most second-level handlers, this is a good practice, because it ensures that you execute the standard behavior before you execute any custom behavior. Next, notice that you call the `PlaySound` Win32 API function to play the sound. You can use this function (defined in the MmSystem unit) to play either WAV files or system sounds, as the SoundB example demonstrates. Here is a textual description of the form of this sample program (from the DFM file):

```
object MdSoundButton1: TMdSoundButton
  Caption = 'Press'
  SoundUp = 'RestoreUp'
  SoundDown = 'RestoreDown'
end
```

NOTE *Selecting a proper value for these sound properties is far from simple. Later in this chapter, I'll show you how to add a property editor to the component to simplify the operation.*

Handling Internal Messages: The Active Button

The Windows interface is evolving toward a new standard, including components that become highlighted as the mouse cursor moves over them. Delphi provides similar support in many of its built-in components. Mimicking this behavior for a button might seem a complex task to accomplish, but it is not. The development of a component can become much simpler once you know which virtual function to override or which message to hook onto.

The next component, the `TMdActiveButton` class, demonstrates this technique by handling some internal Delphi messages to accomplish its task in a simple way. (For information about where these internal Delphi messages come from, see the next section, "Component Messages and Notifications.") The ActiveButton component handles the `cm_MouseEnter` and `cm_MouseExit` internal Delphi messages, which are received when the mouse cursor enters or leaves the area corresponding to the component:

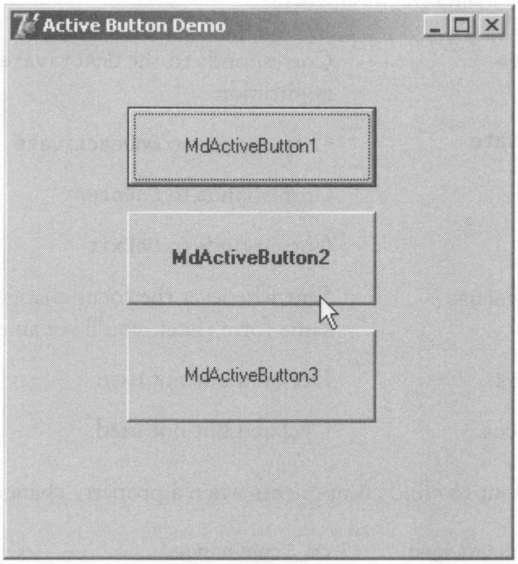
```
type
  TMdActiveButton = class (TButton)
  protected
    procedure MouseEnter (var Msg: TMessage); message cm_mouseEnter;
    procedure MouseLeave (var Msg: TMessage); message cm_mouseLeave;
  end;
```

The code you write for these two methods can do whatever you want. For this example, I've decided to toggle the bold style of the button's font. You can see the effect of moving the mouse over one of these components in Figure 9.9.

```
procedure TMdActiveButton.MouseEnter (var Msg: TMessage);
begin
  Font.Style := Font.Style + [fsBold];
end;

procedure TMdActiveButton.MouseLeave (var Msg: TMessage);
begin
  Font.Style := Font.Style - [fsBold];
end;
```

FIGURE 9.9

An example of the use of the ActiveButton component



You can add other effects, including enlarging the font, making the button the default, or increasing the button's size a little. The best effects usually involve colors, but you must inherit from the `TBitBtn` class to have this support (`TButton` controls have a fixed color).

Component Messages and Notifications

To build the ActiveButton component, I used two internal Delphi component messages, as indicated by their *cm* prefix. These messages can be quite interesting, as the example highlights, but they are almost completely undocumented by Borland. There is also a second group of internal Delphi messages, indicated as component notifications and distinguished by their *cn* prefix. I don't have enough space here to discuss each of them or provide a detailed analysis; browse the VCL source code if you want to learn more.

WARNING *This is a rather advanced topic, so feel free to skip this section if you are new to writing Delphi components. Component messages are not documented in the Delphi help file, so I felt it was important to at least* list *them here.*

COMPONENT MESSAGES

A Delphi component passes *component messages* to other components to indicate any change in its state that might affect those components. Most of these messages begin as Windows messages, but some of them are more complex, higher-level translations and not simple remappings. In addition, components send their own messages as well as forwarding those received from Windows. For example, changing a property value or some other characteristic of the component may necessitate telling one or more other components about the change.

You can group these messages into categories:

- Activation and input focus messages are sent to the component being activated or deactivated, receiving or losing the input focus:

`cm_Activate`	Corresponds to the `OnActivate` event of forms and of the application
`cm_Deactivate`	Corresponds to `OnDeactivate`
`cm_Enter`	Corresponds to `OnEnter`
`cm_Exit`	Corresponds to `OnExit`
`cm_FocusChanged`	Sent whenever the focus changes between components of the same form (later, you'll see an example using this message)
`cm_GotFocus`	Declared but not used
`cm_LostFocus`	Declared but not used

- Messages sent to child components when a property changes:

`cm_BiDiModeChanged`
`cm_BorderChanged`
`cm_ColorChanged`
`cm_Ctl3DChanged`
`cm_CursorChanged`
`cm_EnabledChanged`
`cm_FontChanged`
`cm_IconChanged`
`cm_ShowHintChanged`
`cm_ShowingChanged`
`cm_SysFontChanged`
`cm_TabStopChanged`
`cm_TextChanged`
`cm_VisibleChanged`

Monitoring these messages can help track changes in a property. You might need to respond to these messages in a new component, but it's not likely.

- Messages related to *ParentXxx* properties: `cm_ParentFontChanged`, `cm_ParentColorChanged`, `cm_ParentCtl3DChanged`, `cm_ParentBiDiModeChanged`, and `cm_ParentShowHintChanged`. These are similar to the messages in the previous group.
- Notifications of changes in the Windows system: `cm_SysColorChange`, `cm_WinIniChange`, `cm_TimeChange`, and `cm_FontChange`. Handling these messages is useful only in special components that need to keep track of system colors or fonts.
- Mouse messages: `cm_Drag` is sent many times during dragging operations. `cm_MouseEnter` and `cm_MouseLeave` are sent to the control when the cursor enters or leaves its surface, but they are sent by the `Application` object as low-priority messages. `cm_MouseWheel` corresponds to wheel-based operations.

Application messages:

`cm_AppKeyDown`	Sent to the `Application` object to let it determine whether a key corresponds to a menu shortcut
`cm_AppSysCommand`	Corresponds to the `wm_SysCommand` message
`cm_DialogHandle`	Sent in a DLL to retrieve the value of the `DialogHandle` property (used by some dialog boxes not built with Delphi)
`cm_InvokeHelp`	Sent by code in a DLL to call the `InvokeHelp` method
`cm_WindowHook`	Sent in a DLL to call the `HookMainWindow` and `UnhookMainWindow` methods

You'll rarely need to use these messages. There is also a `cm_HintShowPause` message, which is never handled in VCL.

- Delphi internal messages:

`cm_CancelMode`	Terminates special operations, such as showing the pull-down list of a combo box
`cm_ControlChange`	Sent to each control before adding or removing a child control (handled by some common controls)
`cm_ControlListChange`	Sent to each control before adding or removing a child control (handled by the DBCtrlGrid component)
`cm_DesignHitTest`	Determines whether a mouse operation should go to the component or to the form designer
`cm_HintShow`	Sent to a control just before displaying its hint (only if the `ShowHint` property is True)
`cm_HitTest`	Sent to a control when a parent control is trying to locate a child control at a given mouse position (if any)
`cm_MenuChanged`	Sent after MDI or OLE menu-merging operations

- Messages related to special keys:

`cm_ChildKey`	Sent to the parent control to handle some special keys (in Delphi, this message is handled only by DBCtrlGrid components)
`cm_DialogChar`	Sent to a control to determine whether a given input key is its accelerator character
`cm_DialogKey`	Handled by modal forms and controls that need to perform special actions
`Cm_IsShortCut`	Is currently not used (as most code simply calls `IsShortCut`), but it is intended to be used to identify if a shortcut is known to be supported by a form, through either the `OnShortCut` event, a menu item, or an action.
`cm_WantSpecialKey`	Handled by controls that interpret special keys in an unusual way (for example, using the Tab key for navigation, as some Grid components do)

- Messages for specific components:

`cm_GetDataLink`	Used by DBCtrlGrid controls (and discussed in Chapter 17, "Writing Database Components")
`cm_TabFontChanged`	Used by TabbedNotebook components
`cm_ButtonPressed`	Used by SpeedButtons to notify other sibling SpeedButton components (to enforce radio-button behavior)
`cm_DeferLayout`	Used by DBGrid components

- OLE container messages: `cm_DocWindowActivate`, `cm_IsToolControl`, `cm_Release`, `cm_UIActivate`, and `cm_UIDeactivate`.
- Dock-related messages, including `cm_DockClient`, `cm_DockNotification`, `cmFloat`, and `cm_UndockClient`.
- Method-implementation messages, such as `cm_RecreateWnd`, called inside the `RecreateWnd` method of `TControl`; `cm_Invalidate`, called inside `TControl.Invalidate`; `cm_Changed`, called inside `TControl.Changed`; and `cm_AllChildrenFlipped`, called in the `DoFlipChildren` methods of `TWinControl` and `TScrollingWinControl`. In the similar group fall two action list–related messages, `cm_ActionUpdate` and `cm_ActionExecute`.

COMPONENT NOTIFICATIONS

Component notification messages are those sent from a parent form or component to its children. These notifications correspond to messages sent by Windows to the parent control's window, but logically intended for the control. For example, interaction with controls such as buttons, edit boxes, or list boxes causes Windows to send a `wm_Command` message to the parent of the control. When a Delphi program receives these messages, it forwards the message to the control itself, as a notification.

The Delphi control can handle the message and eventually fire an event. Similar dispatching operations take place for many other messages.

The connection between Windows messages and component notification ones is so tight that you'll often recognize the name of the Windows message from the name of the notification message, replacing the initial *cn* with *wm*. There are several distinct groups of component notification messages:

- General keyboard messages: `cn_Char`, `cn_KeyUp`, `cn_KeyDown`, `cn_SysChar`, and `cn_SysKeyDown`
- Special keyboard messages used only by list boxes with the `lbs_WantKeyboardInput` style: `cn_CharToItem` and `cn_VKeyToItem`
- Messages related to the owner-draw technique: `cn_CompareItem`, `cn_DeleteItem`, `cn_DrawItem`, and `cn_MeasureItem`
- Messages for scrolling, used only by scroll bar and track bar controls: `cn_HScroll` and `cn_VScroll`
- General notification messages, used by most controls: `cn_Command`, `cn_Notify`, and `cn_ParentNotify`
- Control color messages: `cn_CtlColorBtn`, `cn_CtlColorDlg`, `cn_CtlColorEdit`, `cn_CtlColorListbox`, `cn_CtlColorMsgbox`, `cn_CtlColorScrollbar`, and `cn_CtlColorStatic`

Other control notifications are defined for common controls support (in the ComCtrls unit).

AN EXAMPLE OF COMPONENT MESSAGES

As an example of the use of some component messages, I've written the CMNTest program. It has a form with three edit boxes and associated labels. The first message it handles, `cm_DialogKey`, allows it to treat the Enter key as if it were a Tab key. The code of this method checks for the Enter key's code and sends the same message, but passes the `vk_Tab` key code. To halt further processing of the Enter key, you set the result of the message to 1:

```
procedure TForm1.CMDialogKey(var Message: TCMDialogKey);
begin
  if (Message.CharCode = VK_RETURN) then
  begin
    Perform (CM_DialogKey, VK_TAB, 0);
    Message.Result := 1;
  end
  else
    inherited;
end;
```

The second message, `cm_DialogChar`, monitors accelerator keys. This technique can be useful to provide custom shortcuts without defining an extra menu for them. Notice that while this code is correct for a component, in a normal application this can be achieved more easily by handling the form's `OnShortCut` event. In this case, you log the special keys in a label:

```
procedure TForm1.CMDialogChar(var Msg: TCMDialogChar);
begin
  Label1.Caption := Label1.Caption + Char (Msg.CharCode);
  inherited;
end;
```

Finally, the form handles the `cm_FocusChanged` message, to respond to focus changes without having to handle the `OnEnter` event of each of its components. Again, the action displays a description of the focused component:

```
procedure TForm1.CmFocusChanged(var Msg: TCmFocusChanged);
begin
  Label5.Caption := 'Focus on ' + Msg.Sender.Name;
end;
```

The advantage of this approach is that it works independently of the type and number of components you add to the form, and it does so without any special action on your part. Again, this is a trivial example for such an advanced topic, but if you add to this the code of the ActiveButton component, you have at least a few reasons to look into these special, undocumented messages. At times, writing the same code without their support can become extremely complex.

A Dialog Box in a Component

The next component we'll examine is completely different from those you have seen up to now. After building window-based controls and graphic components, I'll now show you how to build a nonvisual component.

The basic idea is that forms are components. When you have built a form that might be particularly useful in multiple projects, you can add it to the Object Repository or make a component out of it. The second approach is more complex than the first, but it makes using the new form easier and allows you to distribute the form without its source code. As an example, I'll build a component based on a custom dialog box, trying to mimic as much as possible the behavior of standard Delphi dialog box components.

The first step in building a dialog box in a component is to write the code for the dialog box itself, using the standard Delphi approach. Just define a new form and work on it as usual. When a component is based on a form, you can almost visually design the component. Of course, once the dialog box has been built, you have to define a component around it in a nonvisual way.

The standard dialog box in this example is based on a list box, because it is common to let a user choose a value from a list of strings. I've customized this common behavior in a dialog box and then used it to build a component. The `ListBoxForm` form has a list box and the typical OK and Cancel buttons, as shown in its textual description:

```
object MdListBoxForm: TMdListBoxForm
  BorderStyle = bsDialog
  Caption = 'ListBoxForm'
  object ListBox1: TListBox
    OnDblClick = ListBox1DblClick
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

The only method of this dialog box form relates to the list box's double-click event, which closes the dialog box as though the user clicked the OK button, by setting the `ModalResult` property of the form to mrOk. Once the form works, you can begin changing its source code, adding the definition of a component and removing the declaration of the global variable for the form.

NOTE *For components based on a form, you can use two Pascal source code files: one for the form and the other for the component encapsulating it. It is also possible to place both the component and the form in a single unit, as I've done for this example. In theory, it would be better to declare the form class in the implementation portion of this unit, hiding it from the component's users. But in practice, this is not a good idea. To manipulate the form visually in the Form Designer, the form class declaration must appear in the interface section of the unit. The rationale behind this behavior of the Delphi IDE is that, among other things, this constraint minimizes the amount of code the module manager has to scan to find the form declaration—an operation that must be performed often to maintain the synchronization of the visual form with the form class definition.*

The most important operation is the definition of the `TMdListBoxDialog` component. This component is defined as nonvisual because its immediate ancestor class is `TComponent`. The component has one public property and these three published properties:

- `Lines` is a `TStrings` object, which is accessed via two methods, `GetLines` and `SetLines`. This second method uses the `Assign` procedure to copy the new values to the private field corresponding to this property. This internal object is initialized in the `Create` constructor and destroyed in the `Destroy` method.
- `Selected` is an integer that directly accesses the corresponding private field. It stores the selected element of the list of strings.
- `Title` is a string used to change the title of the dialog box.

The public property is `SelItem`, a read-only property that automatically retrieves the selected element of the list of strings. Notice that this property has no storage and no data; it accesses other properties, providing a virtual representation of data:

```
type
TMdListBoxDialog = class (TComponent)
private
  FLines: TStrings;
  FSelected: Integer;
  FTitle: string;
  function GetSelItem: string;
  procedure SetLines (Value: TStrings);
  function GetLines: TStrings;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  function Execute: Boolean;
  property SelItem: string read GetSelItem;
published
  property Lines: TStrings read GetLines write SetLines;
  property Selected: Integer read FSelected write FSelected;
  property Title: string read FTitle write FTitle;
end;
```

Most of this example's code is in the `Execute` method, a function that returns True or False depending on the modal result of the dialog box. This is consistent with the `Execute` method of most standard Delphi dialog box components. The `Execute` function creates the form dynamically, sets some of its values using the component's properties, shows the dialog box, and, if the result is correct, updates the current selection:

```
function TMdListBoxDialog.Execute: Boolean;
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create ('No items in the list');
  ListBoxForm := TListBoxForm.Create (Self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
    begin
      Result := True;
      Selected := ListBoxForm.ListBox1.ItemIndex;
    end
    else
      Result := False;
  finally
    ListBoxForm.Free;
  end;
end;
```

Notice that the code is contained within a `try/finally` block, so if a run-time error occurs when the dialog box is displayed, the form will be destroyed anyway. I've also used exceptions to raise an error if the list is empty when a user runs it. This error is by design, and using an exception is a good technique to enforce it. The component's other methods are straightforward. The constructor creates the `FLines` string list, which is deleted by the destructor; the `GetLines` and `SetLines` methods operate on the string list as a whole; and the `GetSelItem` function (which follows) returns the text of the selected item:

```
function TMdListBoxDialog.GetSelItem: string;
begin
  if (Selected >= 0) and (Selected < FLines.Count) then
    Result := FLines [Selected]
  else
    Result := '';
end;
```

Of course, because you are manually writing the component code and adding it to the original form's source code, you have to remember to write the `Register` procedure.

Once you've written the `Register` procedure and the component is ready, you must provide a bitmap. For nonvisual components, bitmaps are very important because they are used not only for the Component Palette, but also when you place the component on a form.

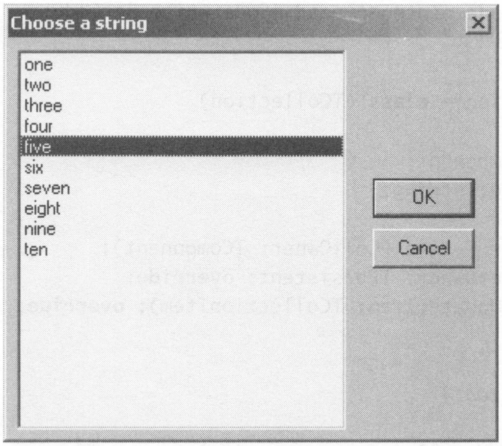
Using the Nonvisual Component

I've written a project to test the component once the bitmap has been prepared and the component has been installed. The form of this test program has a button, an edit box, and the MdListDialog component. In the program, I've added only a few lines of code, corresponding to the button's `OnClick` event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // select the text of the edit, if corresponding to one of the strings
  MdListDialog1.Selected := MdListDialog1.Lines.IndexOf (Edit1.Text);
  // run the dialog and get the result
  if MdListDialog1.Execute then
    Edit1.Text := MdListDialog1.SelItem;
end;
```

That's all the code you need to run the dialog box placed in the component, as you can see in Figure 9.10. As you've seen, this is an interesting approach to the development of some common dialog boxes.

FIGURE 9.10
The ListDialDemo example shows the dialog box encapsulated in the ListDial component.



Collection Properties

At times you need a property holding a list of values, not a single value. Sometimes you can use a `TStringList`-based property, but it accounts only for textual data (even though an object can be attached to each string). When you need a property hosting an array of objects, the most VCL-sound solution is to use a collection. The role of collections, by design, is to build properties that contain a list of values. Examples of Delphi collection properties include the DBGrid component's `Columns` property, and the `TStatusBar` component's `Panels` property.

A collection is basically a container of objects of a given type. For this reason, to define a collection, you need to inherit a new class from the `TCollection` class and also inherit a new class from the `TCollectionItem` class. This second class defines the objects held by the collection; the collection is created by passing to it the class of the objects it will hold.

Not only does the collection class manipulate the items of the collection, but it is also responsible for creating new objects when its `Add` method is called. You cannot create an object and then add it to an existing collection. Listing 9.3 shows two classes for the items and a collection, with their most relevant code.

LISTING 9.3: THE CLASSES FOR A COLLECTION AND ITS ITEMS

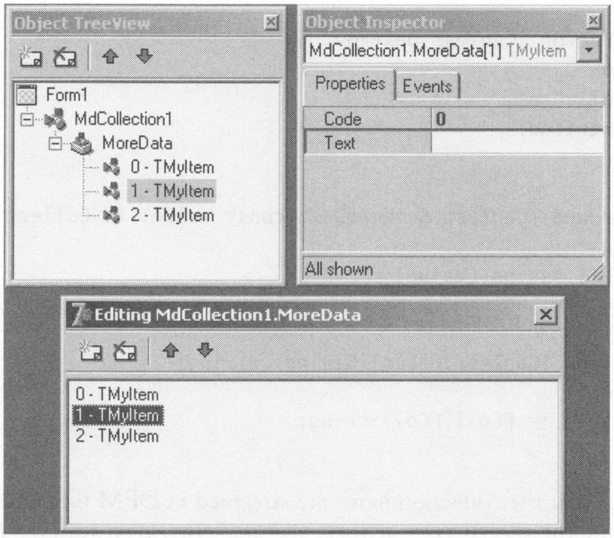
```
type
  TMdMyItem = class (TCollectionItem)
  private
    FCode: Integer;
    FText: string;
    procedure SetCode(const Value: Integer);
    procedure SetText(const Value: string);
  published
    property Text: string read FText write SetText;
    property Code: Integer read FCode write SetCode;
  end;

  TMdMyCollection = class (TCollection)
  private
    FComp: TComponent;
    FCollString: string;
  public
    constructor Create (CollOwner: TComponent);
    function GetOwner: TPersistent; override;
    procedure Update(Item: TCollectionItem); override;
  end;

{ TMdMyCollection }

constructor TMdMyCollection.Create (CollOwner: TComponent);
begin
  inherited Create (TMdMyItem);
  FComp := CollOwner;
end;

function TMdMyCollection.GetOwner: TPersistent;
begin
  Result := FComp;
end;
```

```
procedure TMyCollection.Update(Item: TCollectionItem);
var
  str: string;
  i: Integer;
begin
  inherited;
  // update everything in any case...
  str := '';
  for i := 0 to Count - 1 do
  begin
    str := str + (Items [i] as TMyItem).Text;
    if i < Count - 1 then
      str := str + '-';
  end;
  FCollString := str;
end;
```

The collection must define the `GetOwner` method to be displayed properly in the collection property editor provided by the Delphi IDE. For this reason, it needs a link to the component hosting it, the collection owner (stored in the `FComp` field in the code). You can see this sample component's collection in Figure 9.11.

FIGURE 9.11

The collection editor, with the Object TreeView and the Object Inspector for the collection item


Object TreeView
Form1
MdCollection1
MoreData
0 - TMyItem
1 - TMyItem
2 - TMyItem
Object Inspector
MdCollection1.MoreData[1] TMyItem
Properties
Events
Code
0
Text
All shown
Editing MdCollection1.MoreData
0 - TMyItem
1 - TMyItem
2 - TMyItem

Every time data changes in a collection item, its code calls the `Changed` method (passing True or False to indicate whether the change is local to the item or refers to the entire set of items in the collection). As a result of this call, the `TCollection` class calls the virtual method `Update`, which receives as a parameter the single item requesting the update, or `nil` if all items changed (and when the `Changed` method is called with True as a parameter). You can override this method to update the values of other elements of the collection, of the collection itself, or of the target component.

In this example you update a string with a summary of the collection data that you've added to the collection and that the host component will surface as a property. Using the collection within a component is simple. You declare a collection, create it in the constructor and free it at the end, and expose it through a property:

```
type
  TCanTest = class(TComponent)
  private
    FColl: TMyCollection;
    function GetCollString: string;
  public
    constructor Create (aOwner: TComponent); override;
    destructor Destroy; override;
  published
    property MoreData: TMyCollection read FColl write SetMoreData;
    property CollString: string read GetCollString;
  end;

constructor TCanTest.Create(aOwner: TComponent);
begin
  inherited;
  FColl := TMyCollection.Create (Self);
end;

destructor TCanTest.Destroy;
begin
  FColl.Free;
  inherited;
end;

procedure TCanTest.SetMoreData(const Value: TMyCollection);
begin
  FColl.Assign (Value);
end;

function TCanTest.GetCollString: string;
begin
  Result := FColl.FCollString;
end;
```

Notice that the collection items are streamed in DFM files along with the component hosting them, using the special `item` markers and angle brackets, as in the following example:

```
object MdCollection1: TMdCollection
  MoreData = <
    item
      Text = 'one'
      Code = 1
    end
    item
      Text = 'two'
      Code = 2
    end
    item
      Text = 'three'
      Code = 3
    end>
end
```

Defining Custom Actions

In addition to defining custom components, you can define and register new standard actions, which will be made available in the Action List component's Action Editor. Creating new actions is not complex. You have to inherit from the `TAction` class and override some of the methods of the base class.

You must override three methods:

- The `HandlesTarget` function returns whether the action object wants to handle the operation for the current target, which is by default the control with the focus.
- The `UpdateTarget` procedure can set the user interface of the controls connected with the action, eventually disabling the action if the operation is currently not available.
- You can implement the `ExecuteTarget` method to determine the code to execute, so that the user can select the action and doesn't have to implement it.

To show you this approach in practice, I've implemented the three cut, copy, and paste actions for a list box, in a way similar to what VCL does for an edit box (although I've simplified the code a little). I've written a base class, which inherits from the generic `TListControlAction` class of the ExtActns unit. This base class, `TMdCustomListAction`, adds some common code shared by all the specific actions and publishes a few action properties. The three derived classes have their own `ExecuteTarget` code, plus little more. Here are the four classes:

```
type
  TMdCustomListAction = class (TListControlAction)
  protected
    function TargetList (Target: TObject): TCustomListBox;
    function GetControl (Target: TObject): TCustomListControl;
  public
    procedure UpdateTarget (Target: TObject); override;
  published
    property Caption;
    property Enabled;
    property HelpContext;
    property Hint;
    property ImageIndex;
    property ListControl;
    property ShortCut;
    property SecondaryShortCuts;
    property Visible;
    property OnHint;
  end;

  TMdListCutAction = class (TMdCustomListAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;

  TMdListCopyAction = class (TMdCustomListAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;

  TMdListPasteAction = class (TMdCustomListAction)
  public
    procedure UpdateTarget (Target: TObject); override;
    procedure ExecuteTarget (Target: TObject); override;
  end;
```

The `HandlesTarget` method, one of the three key methods of action classes, is provided by the `TListControlAction` class with this code:

```
function TListControlAction.HandlesTarget(Target: TObject): Boolean;
begin
  Result := ((ListControl <> nil) or
    (ListControl = nil) and (Target is TCustomListControl)) and
    TCustomListControl(Target).Focused;
end;
```

The `UpdateTarget` method has two different implementations. The default implementation is provided by the base class and used by the copy and cut actions. These actions are enabled only if the target list box has at least one item and an item is currently selected. The status of the paste action depends on the Clipboard status:

```
procedure TMdCustomListAction.UpdateTarget (Target: TObject);
begin
  Enabled := (TargetList (Target).Items.Count > 0)
    and (TargetList (Target).ItemIndex >= 0);
end;

function TMdCustomListAction.TargetList (Target: TObject): TCustomListBox;
begin
  Result := GetControl (Target) as TCustomListBox;
end;

function TMdCustomListAction.GetControl(Target: TObject): TCustomListControl;
```

```
begin
  Result := Target as TCustomListControl;
end;

procedure TMdListPasteAction.UpdateTarget (Target: TObject);
begin
  Enabled := Clipboard.HasFormat (CF_TEXT);
end;
```

The `TargetList` function uses the `TListControlAction` class's `GetControl` function, which returns either the list box connected to the action at design time or the target control (the list box control with the input focus).

Finally, the three `ExecuteTarget` methods perform the corresponding actions on the target list box:

```
procedure TMdListCopyAction.ExecuteTarget (Target: TObject);
begin
  with TargetList (Target) do
    Clipboard.AsText := Items [ItemIndex];
end;

procedure TMdListCutAction.ExecuteTarget(Target: TObject);
begin
  with TargetList (Target) do
  begin
    Clipboard.AsText := Items [ItemIndex];
    Items.Delete (ItemIndex);
  end;
end;

procedure TMdListPasteAction.ExecuteTarget(Target: TObject);
begin
  (TargetList (Target)).Items.Add (Clipboard.AsText);
end;
```

Once you've written this code in a unit and added it to a package (in this case, the MdPack package), the final step is to register the new custom actions in a given category. This category is indicated as the first parameter of the `RegisterActions` procedure; the second parameter is the list of action classes to register:

```
procedure Register;
begin
  RegisterActions ('List',
    [TMdListCutAction, TMdListCopyAction, TMdListPasteAction], nil);
end;
```

To test the use of these three custom actions, I've written the ListTest example (included with the source code for this chapter). This program has two list boxes plus a toolbar that contains three buttons connected to the three custom actions and an edit box for entering new values. The program allows a user to cut, copy, and paste list box items. Nothing special, you might think—but the strange fact is that the program has no code!

WARNING *To set up an image for an action (and to define default property values in general) you need to use the third parameter of the* `RegisterActions` *procedure, which is a data module hosting the image list and an action list with the predefined values. As you have to register the actions before you can set up such a data module, you'll need a double registration while developing these actions. This issue is quite complex so I won't cover it here, but a detailed description can be found on* `http://www.blong.com/Conferences/BorCon2002/Actions/2110.htm` *in the sections "Registering Standard Actions" and "Standard Actions And Data Modules."*

Writing Property Editors

Writing components is an effective way to customize Delphi, helping developers to build applications more quickly without requiring a detailed knowledge of low-level techniques. The Delphi environment is also open to extensions. In particular, you can extend the Object Inspector by writing custom property editors and the Form Designer by adding component editors.

Along with these techniques, Delphi offers internal interfaces to add-on tool developers. Using these interfaces, known as the OpenTools API, requires an advanced understanding of how the Delphi environment works and a fairly good knowledge of many advanced techniques that are not discussed in this book. For references to technical information and some examples of these techniques, see Appendix A, "Extra Delphi Tools by the Author."

NOTE *The OpenTools API in Delphi has changed considerably over time. For example, the DsgnIntf unit from Delphi 5 has been split into DesignIntf, DesignEditors, and other specific units. Borland has also introduced interfaces to define the sets of methods for each kind of editor. However, most of the simpler examples, such as those presented in this book, compile almost unchanged from earlier versions of Delphi. For more information, you can study the extensive source code in Delphi's* `\Source\ToolsApi` *directory. Notice also that with Delphi 6 Update Pack 2 Borland has for the first time shipped a Help file with the documentation of the OpenTools API.*

Every property editor must inherit from the abstract `TPropertyEditor` class, which is defined in the DesignEditors unit and provides a standard implementation for the `IProperty` interface. Delphi already defines some specific property editors for strings (the `TStringProperty` class), integers (the `TIntegerProperty` class), characters (the `TCharProperty` class), enumerations (the `TEnumProperty` class), and sets (the `TSetProperty` class), so you can inherit your property editor from the one for the property type you are working with.

In any custom property editor, you must redefine the `GetAttributes` function so it returns a set of values indicating the capabilities of the editor. The most important attributes are paValueList and paDialog. The paValueList attribute indicates that the Object Inspector will show a combo box with a list of values (eventually sorted if the paSortList attribute is set) provided by overriding the `GetValues` method. The paDialog attribute style activates an ellipsis button in the Object Inspector, which executes the editor's `Edit` method.

An Editor for the Sound Properties

The sound button you built earlier has two sound-related properties: `SoundUp` and `SoundDown`. These are strings, so you can display them in the Object Inspector using a default property editor. However, requiring the user to type the name of a system sound or an external file is not friendly, and it's a bit error-prone.

We could write a generic editor to handle filenames, but you want to be able to choose the name of a system sound as well. (System sounds are predefined names of sounds connected with user operations, associated with actual sound files in the Windows Control Panel's Sounds applet.) For this reason, I built a more complex property editor. My editor for sound strings allows a user to either choose a value from a drop-down list or display a dialog box from which to load and test a sound (from a sound file or a system sound). The property editor provides both `Edit` and `GetValues` methods:

```
type
  TSoundProperty = class (TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure Edit; override;
  end;
```

TIP The default Delphi convention is to name a property editor class with a name ending with Property *and all component editors with a name ending with* Editor.

The `GetAttributes` function combines the paValueList (for the drop-down list) and paDialog (for the custom edit box) attributes, and also sorts the lists and allows the selection of the property for multiple components:

```
function TSoundProperty.GetAttributes: TPropertyAttributes;
begin
  // editor, sorted list, multiple selection
  Result := [paDialog, paMultiSelect, paValueList, paSortList];
end;
```

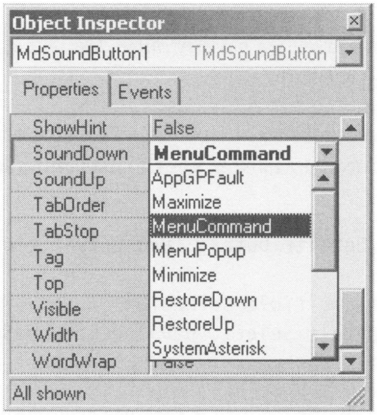
The `GetValues` method calls the procedure it receives as a parameter many times, once for each string it wants to add to the drop-down list (as you can see in Figure 9.12):

```
procedure TSoundProperty.GetValues(Proc: TGetStrProc);
begin
  // provide a list of system sounds
  Proc ('Maximize');
  Proc ('Minimize');
  Proc ('MenuCommand');
  Proc ('MenuPopup');
  Proc ('RestoreDown');
  ...
end;
```

A better approach would be to extract these values from the Windows Registry, where all these names are listed. The `Edit` method is straightforward: It creates and displays a dialog box. I could have displayed the Open dialog box directly, but I decided to add an intermediate step to allow the user to test the sound. This is similar to what Delphi does with graphic properties: You open the preview first, and load the file only after you've confirmed that it's correct. The most important step is to load the file and test it before you apply it to the property. Here is the code for the `Edit` method:

```
procedure TSoundProperty.Edit;
begin
  SoundForm := TSoundForm.Create (Application);
  try
    SoundForm.ComboBox1.Text := GetValue;
    // show the dialog box
    if SoundForm.ShowModal = mrOK then
      SetValue (SoundForm.ComboBox1.Text);
  finally
    SoundForm.Free;
  end;
end;
```

FIGURE 9.12
The list of sounds provides a hint for the user, who can also type in the property value or double-click to activate the editor (shown later, in Figure 9.13).

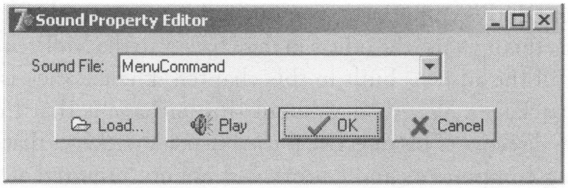


ShowHint False
SoundDown MenuCommand
SoundUp

The `GetValue` and `SetValue` methods are defined by the base class, the string property editor. They read and write the value of the current component's property that you are editing.

As an alternative, you can access the component you're editing by using the `GetComponent` method (which requires a parameter indicating which of the selected components you are working on—0 indicates the first component). When you access the component directly, you also need to call the `Designer` object's `Modified` method (a property of the base class property editor). You don't need this `Modified` call in the example, because the base class `SetValue` method does this automatically for you.

The previous `Edit` method displays a dialog box—a standard Delphi form that is built visually, as always, and added to the package hosting the design-time components. The form is quite simple; a ComboBox displays the values returned by the `GetValues` method, and four buttons allow you to open a file, test the sound, and terminate the dialog box by accepting the values or canceling. You can see an example of the dialog box in Figure 9.13. Providing a drop-down list of values *and* a dialog box for editing a property causes the Object Inspector to display only the arrow button that indicates a drop-down list and to omit the ellipsis button to indicate that a dialog box editor is available. In this case, as it happened for the default `Color` property editor, the dialog box is obtained by double-clicking the current value or pressing Ctrl+Enter.



Sound File: MenuCommand

FIGURE 9.13 The Sound Property Editor's form displays a list of available sounds and lets you load a file and hear the selected sound.

The form's first two buttons have a method assigned to their `OnClick` event:

```
procedure TSoundForm.btnLoadClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    ComboBox1.Text := OpenDialog1.FileName;
end;

procedure TSoundForm.btnPlayClick(Sender: TObject);
begin
  PlaySound (PChar (ComboBox1.Text), 0, snd_Async);
end;
```

Notice that it is far from simple to determine whether a sound is properly defined and is available. (You can check the file, but the system sounds create a few issues.) The `PlaySound` function returns an error code when played synchronously, but only if it can't find the default system sound it attempts to play if it can't find the sound you ask for. If the requested sound is not available, it plays the default system sound and doesn't return the error code. `PlaySound` looks for the sound in the Registry first and, if it doesn't find the sound there, checks to see whether the specified sound file exists.

TIP *If you want to further extend this example, you might add graphics to the drop-down list displayed in the Object Inspector—if you can decide which graphics to attach to particular sounds.*

Installing the Property Editor

After you've written this code, you can install the component and its property editor in Delphi. To accomplish this, you have to add the following statement to the unit's `Register` procedure:

```
procedure Register;
begin
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundUp',
    TSoundProperty);
  RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundDown',
    TSoundProperty);
end;
```

This call registers the editor specified in the last parameter for use with properties of type string (the first parameter), but only for a specific component and for a property with a specific name. These last two values can be omitted to provide more general editors. Registering this editor allows the Object Inspector to show a list of values and the dialog box called by the `Edit` method.

To install this component, you can add its source code file to an existing or new package. Instead of adding this unit and the others in this chapter to the MdPack package, I created a second package containing all the add-ins built in this chapter. The package is named MdDesPk (*Mastering Delphi* design package). What's new about this package is that I compiled it using the `{$DESIGNONLY}` compiler directive. This directive is used to mark packages that interact with the Delphi environment, installing components and editors, but are not required at run time by applications you've built.

NOTE *The source code for all the add-on tools is in the* `MdDesPk` *subdirectory, along with the code for the package used to install them. There are no examples demonstrating how to use these design-time tools, because all you have to do is select the corresponding components in the Delphi environment and see how they behave.*

The property editor's unit uses the SoundB unit, which defines the `TMdSoundButton` component. For this reason, the new package should refer to the existing package. Here is its initial code (I'll add other units to it later in this chapter):

```
package MdDesPk;

{$R *.RES}
{$ALIGN ON}
...
{$DESCRIPTION 'Mastering Delphi DesignTime Package'}
{$DESIGNONLY}

requires
  vcl,
  MdPack,
  designide;

contains
  PeSound in 'PeSound.pas',
  PeFSound in 'PeFSound.pas' {SoundForm};
```

Writing a Component Editor

Using property editors allows the developer to make a component more user-friendly. The Object Inspector represents one of the key pieces of the user interface of the Delphi environment, and Delphi developers use it quite often. However, you can adopt a second approach to customize how a component interacts with Delphi: write a custom component editor.

Just as property editors extend the Object Inspector, component editors extend the Form Designer. When you right-click within a form at design time, you see some default menu items, plus the items added by the component editor of the selected component. Examples of these menu items are those used to activate the Menu Designer, the Fields Editor, the Visual Query Builder, and other editors in the environment. At times, displaying these special editors becomes the default action of a component when it is double-clicked.

Common uses of component editors include adding an About box with information about the developer of the component, adding the component name, and providing specific wizards to set up component properties. In particular, the original intent was to allow a wizard, or some direct code, to set multiple properties in one shot, rather than setting them all individually.

Deriving from the *TComponentEditor* Class

A component editor should generally inherit from the `TComponentEditor` class, which provides the base implementation of the `IComponentEditor` interface. The most important methods of this interface are as follows:

- `GetVerbCount` returns the number of menu items to add to the Form Designer's shortcut menu when the component is selected.
- `GetVerb` is called once for each new menu item and should return the text that will go in the shortcut menu for each.
- `ExecuteVerb` is called when one of the new menu items is selected. The number of the item is passed as the method's parameter.
- `Edit` is called when the user double-clicks the component in the Form Designer to activate the default action.

Once you get used to the idea that a "verb" is nothing but a new menu item with a corresponding action to execute, the names of the methods of this interface become quite intuitive. This interface is much simpler than those of property editors you've seen before.

NOTE *Like property editors, component editors were modified from Delphi 5 to Delphi 6, and are now defined in the DesignEditors and DesignIntf units.*

A Component Editor for the ListDialog

Now that I've introduced the key ideas about writing component editors, let's look at an example—an editor for the ListDialog component built earlier. In my component editor, I want to be able to show an About box, add a copyright notice to the menu (an improper but common use of component editors), and allow users to perform a special action—previewing the dialog box connected with the dialog component. I also want to change the default action to show the About box after a beep (which is not particularly useful but demonstrates the technique).

To implement this component editor, the program must override the four methods listed in the previous section:

```
uses
  DesignIntf;

type
  TMdListCompEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
  end;
```

The first method returns the number of menu items to add to the shortcut menu, in this case 3. This method is called only once: before displaying the menu. The second method is called once for each menu item, so in this case it is called three times:

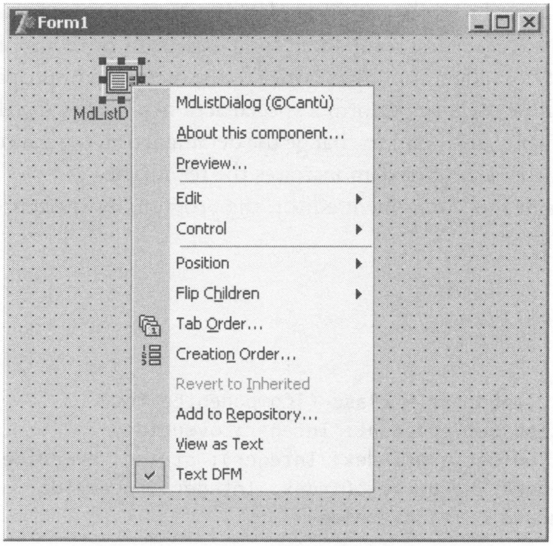
```
function TMdListCompEditor.GetVerb (Index: Integer): string;
begin
  case Index of
    0: Result := ' MdListDialog (©Cantù)';
    1: Result := '&About this component...';
    2: Result := '&Preview...';
  end;
end;
```

This code adds the menu items to the form's shortcut menu, as you can see in Figure 9.14. Selecting any of these menu items activates the `ExecuteVerb` method of the component editor:

```
procedure TMdListCompEditor.ExecuteVerb (Index: Integer);
begin
  case Index of
    0: ; // nothing to do
    1: MessageDlg ('This is a simple component editor'#13 +
      'built by Marco Cantù'#13 +
      'for the book "Mastering Delphi"', mtInformation, [mbOK], 0);
    2: with Component as TMdListDialog do
      Execute;
  end;
end;
```

FIGURE 9.14

The custom menu items added by the component editor of the ListDialog component


Form1
MdListD
MdListDialog (©Cantù)
About this component...
Preview...
Edit
Control
Position
Flip Children
Tab Order...
Creation Order...
Revert to Inherited
Add to Repository...
View as Text
Text DFM

I decided to handle the first two items in a single branch of the `case` statement, although I could have skipped the code for the copyright notice item. The other command changes calls the `Execute` method of the component you are editing, determined using the `TComponentEditor` class's `Component` property. Knowing the type of the component, you can easily access its methods after a dynamic typecast.

The last method refers to the component's default action and is activated by double-clicking the component in the Form Designer:

```
procedure TMdListCompEditor.Edit;
begin
  // produce a beep and show the about box
  Beep;

ExecuteVerb (0);
end;
```

Registering the Component Editor

To make this editor available to the Delphi environment, you need to register it. Once more, you can add to its unit a `Register` procedure and call a specific registration procedure for component editors:

```
procedure Register;
begin
  RegisterComponentEditor (TMdListDialog, TMdListCompEditor);
end;
```

I've added this unit to the MdDesPk package, which includes all the design-time extensions from this chapter. After installing and activating this package, you can create a new project, place a list dialog component in it, and experiment with it.

What's Next?

In this chapter, you have seen how to define various types of properties, how to add events, and how to define and override component methods. You have seen various examples of components, including simple changes to existing components, new graphical components, and, in the final section, a dialog box inside a component. While building these components, you have faced some new Windows programming challenges. In general, programmers often need to use the Windows API directly when writing new Delphi components.

Writing components is a handy technique for reusing software, but to make your components easier to use, you should integrate them as much as possible within the Delphi environment by writing property editors and component editors. You can also write many more extensions of the Delphi IDE, including custom wizards. I've built many Delphi extensions, some of which are discussed in Appendix A.

Chapter 10 focuses on Delphi DLLs. You have used DLLs in previous chapters, and it is time for a detailed discussion of their role and how to build them. I'll also further discuss the use of Delphi packages, which are a special type of DLL. To learn more about component development, refer to Chapter 17, which focuses specifically on data-aware controls and custom dataset components.

Chapter 10

Libraries and Packages

WINDOWS EXECUTABLE FILES COME in two flavors: *programs* (EXEs) and *dynamic link libraries* (DLLs). When you write a Delphi application, you typically generate a program file. However, Delphi applications often use calls to functions stored in dynamic libraries. Each time you call a Windows API function directly, you actually access a dynamic library. It is easy to generate a DLL in the Delphi environment. However, some problems can arise due to the nature of DLLs. Writing a dynamic library in Windows is not always as simple as it seems, because the dynamic library and the calling program need to agree on calling conventions, parameter types, and other details. This chapter covers the basics of DLL programming from the Delphi point of view.

The second part of the chapter will focus on a specific type of dynamic link library: the Delphi *package*. Packages in Delphi provide a good alternative to plain DLLs, although not so many Delphi programmers take advantage of them outside the realm of component writing. Here I'll share with you some tips and techniques for using packages to partition a large application.

The following topics are covered in this chapter:

- Building and using DLLs in Delphi
- Calling DLL functions at run time
- Sharing data in DLLs
- The structure of Delphi packages
- Placing forms in packages

The Role of DLLs in Windows

Before delving into the development of DLLs in Delphi and other programming languages, I'll give you a short technical overview of DLLs in Windows, highlighting the key elements. We will start by looking at dynamic linking, then see how Windows uses DLLs, and end with some general rules to follow when writing DLLs.

What Is Dynamic Linking?

First, it is important to fully understand the difference between static and dynamic linking of functions or procedures. When a subroutine is not directly available in a source file, the compiler adds the subroutine to an internal symbol table. Of course, the Delphi compiler must have seen the declaration of the subroutine and know about its parameters and type, or it will issue an error.

After compilation of a normal—*static*—subroutine, the linker fetches the subroutine's compiled code from a Delphi compiled unit (or static library) and adds it to the program's code. The resulting executable file includes all the code of the program and of the units involved. The Delphi linker is smart enough to include only the minimum amount of code from the program's units and to link only the functions and methods that are actually used. This is why it is called "smart linker."

NOTE *A notable exception to this rule is the inclusion of virtual methods. The compiler cannot determine in advance which virtual methods the program will call, so it has to include them all. For this reason, programs and libraries with too many virtual functions tend to generate larger executable files. While developing the VCL, the Borland developers had to balance the flexibility obtained with virtual functions against the reduced size of the executable files achieved by limiting the virtual functions.*

In the case of dynamic linking, which occurs when your code calls a DLL-based function, the linker uses the information in the `external` declaration of the subroutine to set up an import table in the executable file. When Windows loads the executable file in memory, first it loads all the required DLLs, and then the program starts. During this loading process, Windows fills the program's import table with the addresses of the DLL functions in memory. If for some reason the DLL is not found or a referenced routine is not present in a DLL that is found, the program won't even start.

Each time the program calls an external function, it uses this import table to forward the call to the DLL code (which is now located in the program's address space). Note that this scheme does not involve two different applications. The DLL becomes part of the running program and is loaded in the same address space. All the parameter passing takes place on the application's stack (because the DLL doesn't have a separate stack) or in CPU registers. Because a DLL is loaded into the application's address space, any memory allocations of the DLL or any global data it creates reside in the address space of the main process. Thus, data and memory pointers can be passed directly from the DLL to the program and vice versa. This can also be extended to passing object references, which can be quite troublesome as the EXE and the DLL might have a different compiled class (and you can use packages exactly for this purpose, as we'll see later in this chapter).

There is another approach to using DLLs that is even more dynamic than the one I just discussed: At run time, you can load a DLL in memory, search for a function (provided you know its name), and call the function by name. This approach requires more complex code and takes some extra time to locate the function. The execution of the function, however, occurs with the same speed as calling an implicitly loaded DLL. On the positive side, you don't need to have the DLL available to start the program. We will use this approach in the DynaCall example later in the chapter.

What Are DLLs For?

Now that you have a general idea of how DLLs work, we can focus on the reasons for using them. The first advantage is that if different programs use the same DLL, the DLL is loaded in memory only once, thus saving system memory. DLLs are mapped into the private address space of each process (each running application), but their code is loaded in memory only once.

NOTE *To be more precise, the operating system will try to load the DLL at the same address in each application's address space (using the preferred base address specified by the DLL). If that address is not available in a particular application's virtual address space, the DLL code image for that process will have to be relocated—an operation that is expensive in terms of both performance and memory use, because the relocation happens on a per-process basis, not system-wide.*

Another interesting feature is that you can provide a different version of a DLL, replacing the current one, without having to recompile the application using it. This approach will work, of course, only if the functions in the DLL have the same parameters as the previous version. If the DLL has new functions, it doesn't matter. Problems may arise only if a function in the older version of the DLL is missing in the new one or if a function takes an object reference and the classes, base classes, or even compiler versions don't match.

This second advantage is particularly applicable to complex applications. If you have a very big program that requires frequent updates and bug fixes, dividing it into several executables and dynamic libraries allows you to distribute only the changed portions instead of a single large executable. Doing so makes sense for Windows system libraries in particular: You generally don't need to recompile your code if Microsoft provides an updated version of Windows system libraries—for example, in a new version of the operating system or a service pack.

Another common technique is to use dynamic libraries to store nothing but resources. You can build different versions of a DLL containing strings for different languages and then change the language at run time, or you can prepare a library of icons and bitmaps and then use them in different applications. The development of language-specific versions of a program is particularly important, and Delphi includes support for it through its Integrated Translation Environment (ITE).

Another key advantage is that DLLs are independent of the programming language. Most Windows programming environments, including most macro languages in end-user applications, allow a programmer to call a function stored in a DLL. This flexibility applies only to the use of functions, though. To share objects in a DLL across programming languages, you should move to the COM infrastructure or the .NET architecture.

Rules for Delphi DLL Writers

Delphi DLL programmers need to follow several rules. A DLL function or procedure to be called by external programs must follow these guidelines:

- It must be listed in the DLL's `exports` clause. This makes the routine visible to the outside world.
- Exported functions should also be declared as `stdcall`, to use the standard Win32 parameter-passing technique instead of the optimized `register` parameter-passing technique (which is the default in Delphi). The exception to this rule is if you want to use these libraries only from other Delphi applications. Of course you can also use another calling convention, provided the other compiler understands it (like `cdecl`, which is the default on C compilers).
- The types of a DLL's parameters should be the default Windows types (mostly C-compatible data types), at least if you want to be able to use the DLL within other development environments. There are further rules for exporting strings, as you'll see in the FirstDLL example.

- A DLL can use global data that won't be shared by calling applications. Each time an application loads a DLL, it stores the DLL's global data in its own address space, as you will see in the DllMem example.
- Delphi libraries should trap all internal exceptions, unless you plan to use the library only from other Delphi programs.

Using Existing DLLs

You have already used existing DLLs in examples in this book, when calling Windows API functions. As you might remember, all the API functions are declared in the system Windows unit. Functions are declared in the `interface` portion of the unit, as shown here:

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL; stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints; p4: Integer):
  BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL; stdcall;
```

Then, in the `implementation` portion, instead of providing each function's code, the unit refers to the external definition in a DLL:

```
const
  gdi32 = 'gdi32.dll';

function PlayMetaFile; external gdi32 name 'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name 'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

NOTE `Windows.PAS` *makes heavy use of the* `{$EXTERNALSYM identifier}` *directive. This directive has little to do with Delphi itself; it applies to C++Builder. The symbol prevents the corresponding Delphi symbol from appearing in the C++ translated header file. This action helps keep the Delphi and C++ identifiers in synch, so that code can be shared between the two languages.*

The external definition of these functions refers to the name of the DLL they use. The name of the DLL must include the .DLL extension, or the program will not work under Windows NT/2000/XP (although it will work under Windows 9*x*). The other element is the name of the DLL function. The `name` directive is not necessary if the Delphi function (or procedure) name matches the DLL function name (which is case sensitive).

To call a function that resides in a DLL, you can provide its declaration in the interface section of a unit and external definition in the implementation section, as shown earlier, or you can merge the two in a single declaration in the implementation section of a unit. Once the function is properly defined, you can call it in your Delphi application code just like any other function.

TIP *Delphi includes the Delphi language translation of a large number of Windows APIs, as you can see in the many files available in Delphi's* `Source\Rtl\Win` *folder. More Delphi units referring to other APIs are available as part of the Delphi Jedi project at* `www.delphi-jedi.org`.

Using a C++ DLL

As an example, I've written a DLL in C++ with some trivial functions, just to show you how to call DLLs from a Delphi application. I won't explain the C++ code in detail (it's basically C code) but will focus instead on the calls between the Delphi application and the C++ DLL. In Delphi programming, it is common to use DLLs written in C or C++.

Suppose you are given a DLL built in C or C++. You'll generally have in your hands a .DLL file (the compiled library), an .H file (the declaration of the functions inside the library), and a .LIB file (another version of the list of exported functions for the C/C++ linker). This LIB file is useless in Delphi; the DLL file is used as-is, and the H file must be translated into a Delphi unit with the corresponding declarations.

In the following listing, you can see the declaration of the C++ functions I've used to build the CppDll library example. The complete source code and the compiled version of the C++ DLL and the source code of the Delphi application using it are in the `CppDll` directory. You should be able to compile this code with any C++ compiler; I've tested it only with Borland C++Builder. Here are the C++ declarations of the functions:

```
extern "C" __declspec(dllexport)
int WINAPI Double (int n);
extern "C" __declspec(dllexport)
int WINAPI Triple (int n);
__declspec(dllexport)
int WINAPI Add (int a, int b);
```

The three functions perform some basic calculations on the parameters and return the result. Notice that all the functions are defined with the `WINAPI` modifier, which sets the proper parameter-calling convention; they are preceded by the `__declspec(dllexport)` declaration, which makes the functions available to the outside world.

Two of these C++ functions also use the C naming convention (indicated by the `extern "C"` statement), but the third one, `Add`, doesn't. This difference affects the way you call these functions in Delphi. The internal names of the first two functions correspond to their names in the C++ source code file. But because I didn't use the `extern "C"` clause for the `Add` function, the C++ compiler uses *name mangling*. This technique is used to include information about the number and type of parameters in the function name, which the C++ language requires in order to implement function overloading. The result when using the Borland C++ compiler is a funny function name: `@Add$qqsii`. You must use this name in the Delphi code to call the `Add` DLL function (which explains why you'll generally avoid C++ name mangling in exported functions and declare them all as `extern "C"`). The following are the declarations of the three functions in the Delphi CallCpp example:

```
function Add (A, B: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name '@Add$qqsii';
function Double (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL' name 'Double';
function Triple (N: Integer): Integer;
  stdcall; external 'CPPDLL.DLL';
```

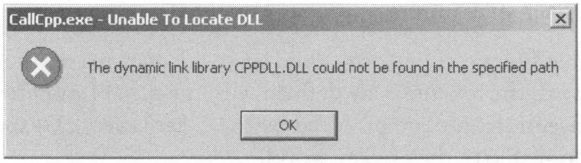
As you can see, you can either provide or omit an alias for an external function. I've provided one for the first function (there was no alternative, because the exported DLL function name `@Add$qqsii` is not a valid Delphi identifier) and for the second, although in the second case it was unnecessary.

If the two names match, you can omit the `name` directive, as I did for the third function. If you are not sure of the actual names of the functions exported by the DLL, you can use Borland's TDump command-line program, available in the Delphi `BIN` folder, using the -ee command-line switch.

Remember to add the `stdcall` directive to each definition, so that the caller module (the application) and the module being called (the DLL) use the same parameter-passing convention. If you fail to do so, you will get unpredictable values passed as parameters, a bug that is very hard to trace.

NOTE *When you have to convert a large C/C++ header file to the corresponding Delphi declarations, instead of doing a manual conversion you can use a tool to partially automate the process. One of these tools is HeadConv, written by Bob Swart. You'll find a copy on his website,* `www.drbob42.com`*. The tool is being extended by Project Jedi, under the name of DARTH project (*`www.delphi-jedi.org/team_darth_home`*). Notice, though, that automatic header translation from C/C++ to Delphi is not possible; the Delphi language is more strongly typed than C/C++, so you have to use types more precisely.*

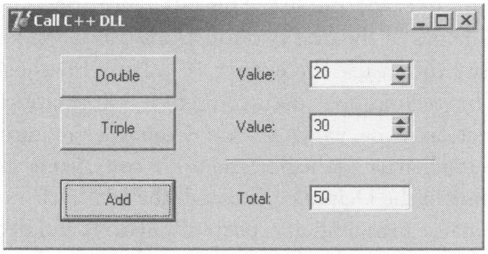
To use this C++ DLL, I've built a Delphi example named CallCpp. Its form has only the buttons used to call the functions of the DLL and some visual components for input and output parameters (see Figure 10.1). Notice that to run this application, you should have the DLL in the same directory as the project, in one of the directories on the path, or in the Windows main folder (`\Windows`, `\WinNT...`) or the Windows' system folder (`\Windows\System`, \WinNT\`System32...`). If you move the executable file to a new directory and try to run it, you'll get a run-time error indicating that the DLL is missing:


CallCpp.exe - Unable To Locate DLL

The dynamic link library CPPDLL.DLL could not be found in the specified path

FIGURE 10.1

The output of the CallCpp example when you have clicked each of the buttons



Value: 20

Value: 30

Total: 50

Creating a DLL in Delphi

Besides using DLLs written in other environments, you can use Delphi to build DLLs that can be used by Delphi programs or with any other development tool that supports DLLs. Building DLLs in Delphi is so easy that you might overuse this feature. In general, I suggest you try to build packages instead of plain DLLs. As I'll discuss later in this chapter, packages often contain components, but they can also include plain noncomponent classes, allowing you to write object-oriented code and to reuse it effectively. Of course, packages can contain also simple routines, constants, variables, etc.

As I've already mentioned, building a DLL is useful when a portion of a program's code is subject to frequent changes. In this case, you can often replace the DLL, keeping the rest of the program unchanged. Similarly, when you need to write a program that provides different features to different groups of users, you can distribute different versions of a DLL to those users.

Your First Delphi DLL

As a starting point in exploring the development of DLLs in Delphi, I'll show you a library built in Delphi. The primary focus of this example will be to show the syntax you use to define a DLL in Delphi, but it will also illustrate a few considerations involved in passing string parameters. To start, select the File ➢ New ➢ Other command and choose the DLL option in the New page of the Object Repository. Doing so creates a very simple source file that begins with the following definition:

```
library Project1;
```

The `library` statement indicates that you want to build a DLL instead of an executable file. Now you can add routines to the library and list them in an `exports` statement:

```
function Triple (N: Integer): Integer; stdcall;
begin
  try
    Result := N * 3;
  except
    Result := -1;
  end;
end;

function Double (N: Integer): Integer; stdcall;
begin
  try
    Result := N * 2;
  except
    Result := -1;
  end;
end;

exports
  Triple, Double;
```

In this basic version of the DLL, you don't need a `uses` statement; but in general, the main project file includes only the `uses` and `exports` statements, whereas the function declarations are placed in a separate unit. In the final source code of the FirstDll example, I've changed the code slightly from the version listed here, to show a message each time a function is called. You can accomplish this two ways; the simplest is to use the Dialogs unit and call the `ShowMessage` function.

The code requires Delphi to link a lot of VCL code into the application. If you statically link the VCL into this DLL, the resulting size will be a few hundred KB. The reason is that the `ShowMessage` function displays a VCL form that contains VCL controls and uses VCL graphics classes; those indirectly refer to things like the VCL streaming system and the VCL application and screen objects.

In this case, a better alternative is to show the messages using direct API calls, using the Windows unit and calling the `MessageBox` function, so that the VCL code is not required. This code change brings the size of the application down to less than 50 KB.

NOTE *This huge difference in size underlines the fact that you should not overuse DLLs in Delphi, to avoid compiling the VCL code in multiple executable files. Of course, you can reduce the size of a Delphi DLL by using run-time packages, as detailed later in this chapter.*

If you run a test program like the CallFrst example (described later) using the API-based version of the DLL, its behavior won't be correct. In fact, you can click the buttons that call the DLL functions several times without first closing the message boxes displayed by the DLL. This happens because the first parameter of the `MessageBox` API call is zero. Its value should instead be the handle of the program's main form or the application form—information you don't have at hand in the DLL.

OVERLOADED FUNCTIONS IN DELPHI DLLS

When you create a DLL in C++, overloaded functions use name mangling to generate a different name for each function. The type of the parameters is included right in the name, as you saw in the CppDll example.

When you create a DLL in Delphi and use overloaded functions (that is, multiple functions using the same name and marked with the `overload` directive), Delphi allows you to export only one of the overloaded functions with the original name, indicating its parameters list in the `exports` clause. If you want to export multiple overloaded functions, you should specify different names in the `exports` clause to distinguish the overloads. This technique is demonstrated by this portion of the FirstDLL code:

```
function Triple (C: Char): Integer; stdcall; overload;
function Triple (N: Integer): Integer; stdcall; overload;

exports
  Triple (N: Integer),
  Triple (C: Char) name 'TripleChar';
```

NOTE *The reverse is possible as well: You can import a series of similar functions from a DLL and define them all as overloaded functions in the Delphi declaration. Delphi's* `OpenGL.PAS` *unit contains a series of examples of this technique.*

EXPORTING STRINGS FROM A DLL

In general, functions in a DLL can use any type of parameter and return any type of value. There are two exceptions to this rule:

- If you plan to call the DLL from other programming languages, you should try using Windows native data types instead of Delphi-specific types. For example, to express color values, you should use integers or the Windows `ColorRef` type instead of the Delphi native `TColor` type, doing the appropriate conversions (as in the FormDLL example, described in the next section). For compatibility, you should avoid using some other Delphi types, including objects (which cannot be used by other languages) and Delphi strings (which can be replaced by `PChar` strings). In other words, every Windows development environment must support the basic types of the API, and if you stick to them, your DLL will be usable with other development environments. Also, Delphi file variables (text files and binary file of record) should not be passed out of DLLs, but you can use Win32 file handles.

- Even if you plan to use the DLL only from a Delphi application, you cannot pass Delphi strings (and dynamic arrays) across the DLL boundary without taking some precautions. This is the case because of the way Delphi manages strings in memory—allocating, reallocating, and freeing them automatically. The solution to the problem is to include the ShareMem system unit both in the DLL and in the program using it. This unit must be included as the first unit of each of the projects. Moreover, you have to deploy the `BorlndMM.DLL` file (the name stands for Borland Memory Manager) along with the program and the specific library.

In the FirstDLL example, I've included both approaches: One function receives and returns a Delphi string, and another receives as parameter a `PChar` pointer, which is then filled by the function. The first function is written as usual in Delphi:

```
function DoubleString (S: string; Separator: Char): string; stdcall;
begin
  try
    Result := S + Separator + S;
  except
    Result := '[error]';
  end;
end;
```

The second function is quite complex because `PChar` strings don't have a simple `+` operator, and they are not directly compatible with characters; the separator must be turned into a string before being adding. Here is the complete code; it uses input and output `PChar` buffers, which are compatible with any Windows development environment:

```
function DoublePChar (BufferIn, BufferOut: PChar;
  BufferOutLen: Cardinal; Separator: Char): LongBool; stdcall;
var
  SepStr: array [0..1] of Char;
begin
  try
    // if the buffer is large enough
    if BufferOutLen > StrLen (BufferIn) * 2 + 2 then
    begin
      // copy the input buffer in the output buffer
      StrCopy (BufferOut, BufferIn);
      // build the separator string (value plus null terminator)
      SepStr [0] := Separator;
      SepStr [1] := #0;
      // append the separator
      StrCat (BufferOut, SepStr);
```

```
        // append the input buffer once more
        StrCat (BufferOut, BufferIn);
        Result := True;
      end
      else
        // not enough space
        Result := False;
    except
      Result := False;
    end;
  end;
```

This second version of the code is certainly more complex, but the first can be used only from Delphi. Moreover, the first version requires you to include the ShareMem unit and to deploy the file `BorlndMM.DLL`, as discussed earlier.

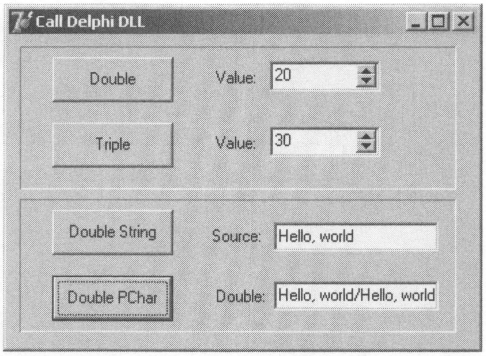
Calling the Delphi DLL

How can you use the library you've just built? You can call it from within another Delphi project or from other environments. As an example, I've built the CallFrst project (stored in the `FirstDLL` directory). To access the DLL functions, you must declare them as external, as with the C++ DLL. This time, however, you can copy and paste the definition of the functions from the source code of the Delphi DLL, adding the `external` clause, as follows:

```
function Double (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';
```

This declaration is similar to those used to call the C++ DLL. This time, however, you have no problems with function names. Once they are redeclared as external, the functions of the DLL can be used as if they were local functions. Here are two examples, with calls to the string-related functions (an example of the output is visible in Figure 10.2):

```
procedure TForm1.BtnDoubleStringClick(Sender: TObject);
begin
  // call the DLL function directly
  EditDouble.Text := DoubleString (EditSource.Text, ';');
end;

procedure TForm1.BtnDoublePCharClick(Sender: TObject);
var
  Buffer: string;
begin
  // make the buffer large enough
  SetLength (Buffer, 1000);
  // call the DLL function
  if DoublePChar (PChar (EditSource.Text), PChar (Buffer), 1000, '/') then
    EditDouble.Text := Buffer;
end;
```



Value: 20

Value: 30

Source: Hello, world

Double: Hello, world/Hello, world

FIGURE 10.2
The output of the CallFrst example, which calls the DLL you've built in Delphi

Advanced Features of Delphi DLLs

Beside this introductory example, you can do a few extra things with dynamic libraries in Delphi. You can use some new compiler directives to affect the name of the library, you can call a DLL at run time, and you can place entire Delphi forms inside a dynamic library. These are the topics of the following sections.

Changing Project and Library Names

For a library, as for a standard application, you end up with a library name matching a Delphi project filename. Following a technique similar to that introduced in Kylix for compatibility with standard Linux naming conventions for shared object libraries (the Linux equivalent of Windows DLLs), Delphi 6 introduced special compiler directives you can use in libraries to determine their executable filename. Some of these directives make more sense in the Linux world than on Windows, but they've all been added anyway:

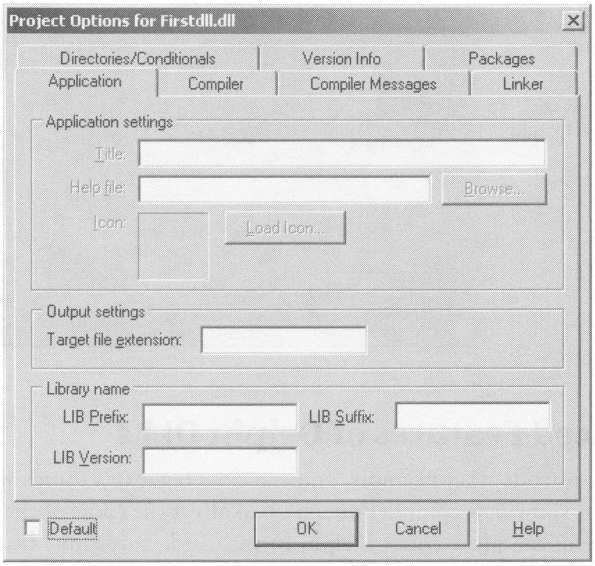
- `$LIBPREFIX` is used to add something in front of the library name. Paralleling the Linux technique of adding *lib* in front of library names, this directive is used by Kylix to add *bpl* at the beginning of package names. It is necessary because Linux uses a single extension (.SO) for libraries, whereas in Windows you can have different library extensions, something Borland uses for packages (.BPL).
- `$LIBSUFFIX` is used to add text after the library name and before the extension. This text can be used to specify versioning information or other variations on the library name and can be quite useful on Windows.
- `$LIBVERSION` is used to add a version number after the extension—something very common in Linux, but that you should generally avoid on Windows.

These directives can be set in the IDE from the Application page of the Project Options dialog box, as you can see in Figure 10.3. As an example, consider the following directives, which generate a library called `MarcoNameTest60.dll`:

```
library NameTest;
{$LIBPREFIX 'Marco'}
{$LIBSUFFIX '60'}
```

FIGURE 10.3

The Application page of the Project Options dialog box now has a Library Name section.


Project Options for Firstdll.dll
Directories/Conditionals
Version Info
Packages
Application
Compiler
Compiler Messages
Linker
Application settings
Title:
Help file:
Browse...
Icon:
Load Icon...
Output settings
Target file extension:
Library name
LIB Prefix:
LIB Suffix:
LIB Version:
Default
OK
Cancel
Help

NOTE *Delphi 6 packages introduced the extensive use of the* `$LIBSUFFIX` *directive. For this reason, the VCL package now generates the* `VCL.DCP` *file and the* `VCL70.BPL` *file. The advantage of this approach is that you won't need to change the* `requires` *portions of your packages for every new version of Delphi. Of course, this is helpful when you move projects from Delphi 6 to Delphi 7, because past versions of Delphi didn't provide this feature. When you reopen Delphi 5 packages you still have to* upgrade *their source code, an operation the Delphi IDE does automatically for you.*

Calling a DLL Function at Run Time

Up to now, you've referenced in your code the functions exported by the libraries, so the DLLs were loaded along with the program. I mentioned earlier that you can also delay the loading of a DLL until the moment it is needed, so you can use the rest of the program in case the DLL is not available.

Dynamic loading of a DLL in Windows is accomplished by calling the `LoadLibrary` API function, which searches for the DLL in the program folder, in the folders on the path, and in some system folders. If the DLL is not found, Windows will show an error message, something you can skip by calling Delphi's `SafeLoadLibrary` function. This function has the same effect as the API it encapsulates, but it suppresses the standard Windows error message and should be the preferred way to load libraries dynamically in Delphi.

If the library is found and loaded (something you know by checking the return value of `LoadLibrary` or `SafeLoadLibrary`), a program can call the `GetProcAddress` API function, which searches the DLL's exports table, looking for the name of the function passed as a parameter. If `GetProcAddress` finds a match, it returns a pointer to the requested procedure. Now you can cast this function pointer to the proper data type and call it.

Whichever loading functions you've used, don't forget to call `FreeLibrary` at the end, so that the DLL can be properly released from memory. In fact, the system uses a reference-counting technique for libraries, releasing them when each loading request has been followed by a freeing request.

The example I've built to show dynamic DLL loading is named DynaCall. It uses the FirstDLL library built earlier in this chapter (to make the program work, you have to copy the DLL from its source folder into the folder as the DynaCall example). Instead of declaring the `Double` and `Triple` functions and using them directly, this example obtains the same effect with somewhat more complex code. The advantage, however, is that the program will run even without the DLL. Also, if new *compatible* functions are added to the DLL, you won't have to revise the program's source code and recompile it to access those new functions. Here is the core code of the program:

```
type
  TIntFunction = function (I: Integer): Integer; stdcall;

const
  DllName = 'Firstdll.dll';

procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFunct: TIntFunction;
begin
  HInst := SafeLoadLibrary (DllName);
  if HInst > 0 then
  try
    FPointer := GetProcAddress (HInst,
      PChar (Edit1.Text));
    if FPointer <> nil then
    begin
      MyFunct := TIntFunction (FPointer);
      SpinEdit1.Value := MyFunct (SpinEdit1.Value);
    end
    else
      ShowMessage (Edit1.Text + ' DLL function not found');
  finally
    FreeLibrary (HInst);
  end
  else
    ShowMessage (DllName + ' library not found');
end;
```

WARNING *As the library uses the Borland memory manager, the program dynamically loading it must do the same. So you need to add the ShareMem unit in the project of the DynaCall example. Oddly enough, this was not so with past versions of Delphi, in case the library didn't effectively use strings. Be warned that if you omit this inclusion, you'll get a harsh system error, which can even stall the debugger on the* `FreeLIrbary` *call.*

How do you call a procedure in Delphi, once you have a pointer to it? One solution is to convert the pointer to a procedural type and then call the procedure using the procedural-type variable, as in the previous listing. Notice that the procedural type you define must be compatible with the definition of the procedure in the DLL. This is the Achilles' heel of this method—there is no actual check of the parameter types.

What is the advantage of this approach? In theory, you can use it to access any function of any DLL at any time. In practice, it is useful when you have different DLLs with compatible functions or a single DLL with several compatible functions, as in this case. You can call the `Double` and `Triple` methods by entering their names in the edit box. Now, if someone gives you a DLL with a new function receiving an integer as a parameter and returning an integer, you can call it by entering its name in the edit box. You don't even need to recompile the application.

With this code, the compiler and the linker ignore the existence of the DLL. When the program is loaded, the DLL is not loaded immediately. You might make the program even more flexible and let the user enter the name of the DLL to use. In some cases, this is a great advantage. A program may switch DLLs at run time, something the direct approach does not allow. Note that this approach to loading DLL functions is common in macro languages and is used by many visual programming environments.

Only a system based on a compiler and a linker, such as Delphi, can use the direct approach, which is generally more reliable and also a little faster. In my opinion, the indirect loading approach of the DynaCall example is useful only in special cases, but it can be extremely powerful. On the other hand, I see a lot of value in using dynamic loading for packages including forms, as you'll see toward the end of this chapter.

Placing Delphi Forms in a Library

Besides writing a library with functions and procedures, you can place a complete form built with Delphi into a dynamic library. This can be a dialog box or any other kind of form, and it can be used not only by other Delphi programs, but also by other development environments or macro languages with the ability to use dynamic link libraries. Once you've created a new library project, all you need to do is add one or more forms to the project and then write exported functions that will create and use those forms.

For example, a function activating a modal dialog box to select a color could be written like this:

```
function GetColor (Col: LongInt): LongInt; cdecl;
var
  FormScroll: TFormScroll;
begin
  // default value
  Result := Col;
  try
    FormScroll := TFormScroll.Create (Application);
    try
      // initialize the data
      FormScroll.SelectedColor := Col;
      // show the form
      if FormScroll.ShowModal = mrOK then
        Result := FormScroll.SelectedColor;
    finally
      FormScroll.Free;
    end;
  except
    on E: Exception do
```

```
      MessageDlg ('Error in library: ' + E.Message, mtError, [mbOK], 0);
    end;
  end;
```

What makes this different from the code you generally write in a program is the use of exception handling:

- A `try/except` block protects the whole function. Any exception generated by the function will be trapped, and an appropriate message will be displayed. You handle every possible exception because the calling application might be written in any language—in particular, one that doesn't know how to handle exceptions. Even when the caller is a Delphi program, it is sometimes helpful to use the same protective approach.
- A `try/finally` block protects the operations on the form, ensuring that the form object will be properly destroyed even when an exception is raised.

By checking the return value of the `ShowModal` method, the program determines the result of the function. I've set the default value before entering the `try` block to ensure that it will always be executed (and also to avoid the compiler warning indicating that the result of the function might be undefined).

You can find this code snippet in the FormDLL and UseCol projects, available in the `FormDLL` folder. (There's also a `WORDCALL.TXT` file showing how to call the routine from a Word macro.). The example also shows that you can add a modeless form to the DLL, but doing so causes far too much trouble. The modeless form and the main form are not synchronized, because the DLL has its own global `Application` object in its own copy of the VCL. This situation can be partially fixed by copying the `Handle` of the application's `Application` object to the `Handle` of the library's `Application` object. Not all of the problems are solved with the code that you can find in the example. A better solution might be to compile the program and the library to use Delphi packages, so that the VCL code and data won't be duplicated. But this approach still causes a few troubles: it's generally advised that you don't use Delphi DLLs and packages together. So what is the best suggestion I can give you? For making the forms of a library available to other Delphi programs, use packages instead of plain DLLs!

Libraries in Memory: Code and Data

Before I discuss packages, I want to focus on a technical element of dynamic libraries: how they use memory. Let's start with the code portion of the library, then we'll focus on its global data. When Windows loads the code of a library, like any other code module, it has to do a *fixup* operation. This *fixup* consists of patching addresses of jumps and internal function calls with the actual memory address where they've been loaded. The effect of this operation is that the code-loaded memory depends on where it has been loaded.

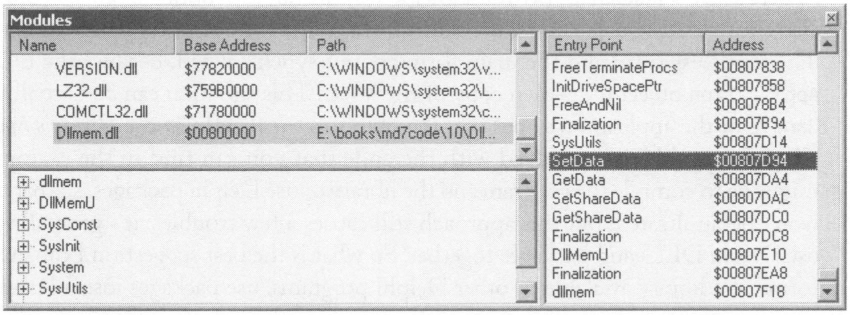
This is not an issue for executable files, but might cause a significant problem for libraries. If two executables load the same library at the same base address, there will be only one physical copy of the DLL code in the RAM (the physical memory) of the machine, thus saving memory space. If the second time the library is loaded the memory address is already in use, it needs to be *relocated*, that is, moved with a different *fixup* applied. So you'll end up with a second physical copy of the DLL code in RAM.

You can use the dynamic loading technique, based on the `GetProcAddress` API function, to test which memory address of the current process a function has been mapped to. The code is as follows:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  HDLLInst: THandle;
begin
  HDLLInst := SafeLoadLibrary ('dllmem');
  Label1.Caption := Format ('Address: %p', [
    GetProcAddress (HDLLInst, 'SetData')]);
  FreeLibrary (HDLLInst);
end;
```

This code displays, in a label, the memory address of the function, within the address space of the calling application. If you run two programs using this code, they'll generally both show the same address. This technique demonstrates that the code is loaded only once at a common memory address.

Another technique to get more information about what's going on is to use Delphi's Modules window, which shows the base address of each library referenced by the module and the address of each function within the library, as shown here:



Name	Base Address	Path
VERSION.dll	$77820000	C:\WINDOWS\system32\v...
LZ32.dll	$759B0000	C:\WINDOWS\system32\L...
COMCTL32.dll	$71780000	C:\WINDOWS\system32\c...
Dllmem.dll	$00800000	E:\books\md7code\10\Dll...

Entry Point	Address
FreeTerminateProcs	$00807838
InitDriveSpacePtr	$00807858
FreeAndNil	$008078B4
Finalization	$00807B94
SysUtils	$00807D14
SetData	$00807D94
GetData	$00807DA4
SetShareData	$00807DAC
GetShareData	$00807DC0
Finalization	$00807DC8
DllMemU	$00807E10
Finalization	$00807EA8
dllmem	$00807F18

It's important to know that the base address of a DLL is something you can request by setting the base address option. In Delphi this address is determined by the Image Base value in the linker page of the Project Options dialog box. In the DllMem library, for example, I've set it to `$00800000`. You need to have a different value for each of your libraries, verifying that it doesn't clash with any system library or other library (package, ActiveX, and so on) used by the executable. Again, this is something you can figure out using the Module window of the debugger.

Although this doesn't guarantee a unique placement, setting a base address for the library is always better than not setting one; in this case a relocation always takes place, but the chance that two different executables will relocate the same library at the same address are not high.

NOTE *You can also use Process Explorer from* `http://www.sysinternals.com` *to examine any process on any machine. This tool even has an option to highlight relocated DLLs. Check the effect of running the same program with its libraries on different operating systems (Windows 2000, Windows XP, and Windows ME) and settle on an unused area.*

This is the case for the DLL code, but what about the global data? Basically, each copy of the DLL has its own copy of the data, in the address space of the calling application. However, it is possible to share global data between applications using a DLL. The most common technique for sharing data is to use memory-mapped files. I'll use this technique for a DLL, but it can also be used to share data directly among applications.

This example is called DllMem for the library and UseMem for the demo application. The DLL code has a project file that exports four subroutines:

```
library dllmem;

uses
  SysUtils,
  DllMemU in 'DllMemU.pas';

exports
  SetData, GetData,
  GetShareData, SetShareData;
end.
```

The actual code is in the secondary unit (`DllMemU.PAS`), which contains the code for the four routines that read or write two global memory locations. These memory locations hold an integer and a pointer to an integer. Here are the variable declarations and the two Set routines:

```
var
  PlainData: Integer = 0; // not shared
  ShareData: ^Integer; // shared

procedure SetData (I: Integer); stdcall;
begin
  PlainData := I;
end;

procedure SetShareData (I: Integer); stdcall;
begin
  ShareData^ := I;
end;
```

Sharing Data with Memory-Mapped Files

For the data that isn't shared, there isn't anything else to do. To access the shared data, however, the DLL has to create a memory-mapped file and then get a pointer to this memory area. These operations require two Windows API calls:

- `CreateFileMapping` requires as parameters the filename (or `$FFFFFFFF` to use a virtual file in memory), some security and protection attributes, the size of the data, and an internal name (which must be the same to share the mapped file from multiple calling applications).
- `MapViewOfFile` requires as parameters the handle of the memory-mapped file, some attributes and offsets, and the size of the data (again).

Here is the source code of the `initialization` section, which is executed every time the DLL is loaded into a new process space (that is, once for each application that uses the DLL):

```
var
  hMapFile: THandle;

const
  VirtualFileName = 'ShareDllData';
  DataSize = sizeof (Integer);

initialization
  // create memory mapped file
  hMapFile := CreateFileMapping ($FFFFFFFF, nil,
    Page_ReadWrite, 0, DataSize, VirtualFileName);
  if hMapFile = 0 then
    raise Exception.Create ('Error creating memory-mapped file');

  // get the pointer to the actual data
  ShareData := MapViewOfFile (
    hMapFile, File_Map_Write, 0, 0, DataSize);
```

When the application terminates and the DLL is released, it has to free the pointer to the mapped file and the file mapping:

```
finalization
  UnmapViewOfFile (ShareData);
  CloseHandle (hMapFile);
```

The UseMem demo program's form has four edit boxes (two with an UpDown control connected), five buttons, and a label. The first button saves the value of the first edit box in the DLL data, getting the value from the connected UpDown control:

```
SetData (UpDown1.Position);
```

If you click the second button, the program copies the DLL data to the second edit box:

```
Edit2.Text := IntToStr(GetData);
```

The third button is used to display the memory address of a function, with the source code shown at the beginning of this section. The last two buttons have basically the same code as the first two, but they call the `SetShareData` procedure and the `GetShareData` function.

If you run two copies of this program, you can see that each copy has its own value for the plain global data of the DLL, whereas the value of the shared data is common. Set different values in the two programs and then get them in both, and you'll see what I mean. This situation is illustrated in Figure 10.4.

WARNING *Memory-mapped files reserve a minimum of a 64 KB range of virtual addresses and consume physical memory in 4 KB pages. The example's use of 4-byte Integer data in shared memory is rather expensive, especially if you use the same approach for sharing multiple values. If you need to share several variables, you should place them all in a single shared memory area (accessing the different variables using pointers or building a record structure for all of them).*

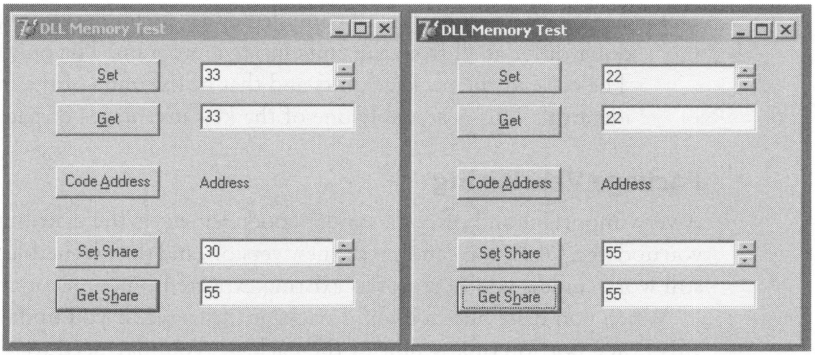


FIGURE 10.4
If you run two copies of the UseMem program, you'll see that the global data in its DLL is not shared.

Using Delphi Packages

In Delphi, component packages are an important type of DLL. Packages allow you to bundle a group of components and then link the components either statically (adding their compiled code to the executable file of your application) or dynamically (keeping the component code in a DLL, the run-time package that you'll distribute along with your program, along with all other packages you will need). In Chapter 9, "Writing Delphi Components," you saw how to build a package. Now I want to underline some advantages and disadvantages of the two forms of linking for a package. You need to keep many elements in mind:

- Using a package as a DLL makes the executable files much smaller.
- Linking the package units into the program allows you to distribute only part of the package code. The size of the executable file of an application plus the size of the required package DLLs is always much bigger than the size of the statically linked program. The linker includes only the code used by the program, whereas a package must link in all the functions and classes declared in the `interface` sections of all the units contained in the package.
- If you distribute several Delphi applications based on the same packages, you might end up distributing less code, because the run-time packages are shared. In other words, once the users of your application have the standard Delphi run-time packages, you can ship them very small programs.
- If you run several Delphi applications based on the same packages, you can save some memory space at run time; the code of the run-time packages is loaded in memory only once among the multiple Delphi applications.
- Don't worry too much about distributing a large executable file. Keep in mind that when you make minor changes to a program, you can use any of various tools to create a *patch file*, so that you distribute only a file containing the differences, not a complete copy of the files.
- If you place a few of your program's forms in a run-time package, you can share them among programs. When you modify these forms, however, you'll generally need to recompile the main program as well, and distribute both of them again to your users. The next section discusses this complex topic in detail.

- A package is a collection of compiled units (including classes, types, variables, routines), which don't differ at all from the units inside a program. The only difference is in the build process. The code of the package units and that of the units of the main program using them remains identical. This is arguably one of the key advantages of packages over DLLs.

Package Versioning

A very important and often misunderstood element is the distribution of updated packages. When you update a DLL, you can ship the new version, and the executable programs requiring this DLL will still work (unless you've removed existing exported functions or changed some of their parameters).

When you distribute a Delphi package, however, if you update the package and modify the `interface` portion of any unit of the package, you may need to recompile all the applications that use the package. This step is required if you add methods or properties to a class, but not if you add new global symbols (or modify anything not used by client applications). There is no problem if you make changes affecting only the `implementation` section of the package's units.

A DCU file in Delphi has a version tag based on its timestamp and a checksum computed from the interface portion of the unit. When you change the `interface` portion of a unit, every other unit based on it should be recompiled. The compiler compares the timestamp and checksum of the unit from previous compilations with the new timestamp and checksum, and decides whether the dependent unit must be recompiled. For this reason, you must recompile each unit when you get a new version of Delphi that has modified system units.

In Delphi 3 (when packages were first introduced), the compiler added an extra entry function to the package library named with a checksum of the package, obtained from the checksum of the units it contained and the checksum of the packages it required. This checksum function was then called by programs using the package so that an older executable would fail at startup.

Delphi 4 and following versions up to Delphi 7 have relaxed the run-time constraints of the package. (The design-time constraints on DCU files remain identical, though.) The checksum of the packages is no longer checked, so you can directly modify the units that are part of a package and deploy a new version of the package to be used with the existing executable file. Because methods are referenced by name, you cannot remove any existing method. You cannot even change its parameters, because of name-mangling techniques that protect a package's method against changes in parameters.

Removing a method referenced from the calling program will stop the program during the loading process. If you make other changes, however, the program might fail unexpectedly during its execution. For example, if you replace a component placed on a form compiled in a package with a similar component, the calling program might still be able to access the component in that memory location, although it is now different!

If you decide to follow this treacherous road of changing the interface of units in a package without recompiling all the programs that use it, you should at least limit your changes. When you add new properties or nonvirtual methods to the form, you should be able to maintain full compatibility with existing programs already using the package. Also, adding fields and virtual methods might affect the internal structure of the class, leading to problems with existing programs that expect a different class data and virtual method table (VMT) layout.

WARNING *Here I'm referring to the distribution of compiled programs divided between EXEs and packages, not to the distribution of components to other Delphi developers. In this latter case the versioning rules are more stringent, and you must take extra care in package versioning.*

Having said this, I recommend never changing the interface of any unit exported by your packages. To accomplish this, you can add to your package a unit with form-creation functions (as in the DLL with forms presented earlier) and use it to access another unit, which defines the form. Although there is no way to *hide* a unit that is linked into a package, if you never directly use the class defined in a unit, but use it only through other routines, you'll have more flexibility in modifying it. You can also use form inheritance to modify a form within a package without affecting the original version.

The most stringent rule for packages is the following one used by component writers: For long-term deployment and maintenance of code in packages, plan on having a major release with minor maintenance releases. A major release of your package will require all client programs to be recompiled from source; the package file should be renamed with a new version number, and the interface sections of units can be modified. Maintenance releases of that package should be restricted to implementation changes to preserve full compatibility with existing executables and units, as is generally done by Borland with its Update Packs.

Forms Inside Packages

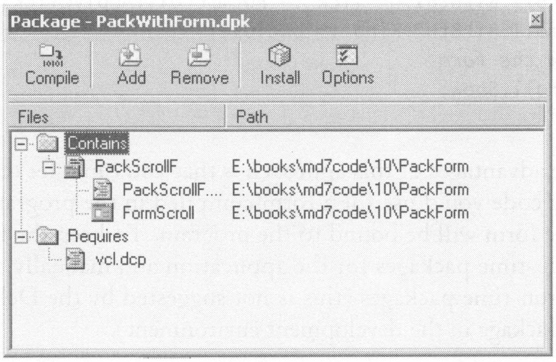
In Chapter 9, I discussed the use of component packages in Delphi applications. Now I'm discussing the use of packages and DLLs for partitioning an application, so I'll begin talking about the development of packages holding forms. I've mentioned earlier in this chapter that you can use forms inside DLLs, but doing so causes quite a few problems. If you are building both the library and the executable file in Delphi, using packages results in a much better and cleaner solution.

At first sight, you might believe that Delphi packages are solely a way to distribute components to be installed in the environment. However, you can also use packages as a way to structure your code but, unlike DLLs, retain the full power of Delphi's OOP. Consider this: A package is a collection of compiled units, and your program uses several units. The units the program refers to will be compiled inside the executable file, unless you ask Delphi to place them inside a package. As discussed earlier, this is one of the main reasons for using packages.

To set up an application so that its code is split among one or more packages and a main executable file, you only need to compile some of the units in a package and then set up the options of the main program to dynamically link this package. For example, I made a copy of the "usual" color selection form and renamed its unit PackScrollF; then I created a new package and added the unit to it, as you can see in Figure 10.5.

FIGURE 10.5

The structure of the package hosting a form in Delphi's Package Editor



Files	Path
Contains	
PackScrollF	E:\books\md7code\10\PackForm
PackScrollF....	E:\books\md7code\10\PackForm
FormScroll	E:\books\md7code\10\PackForm
Requires	
vcl.dcp	

Before compiling this package, you should change its default output directories to refer to the current folder, not the standard `/Projects/Bpl` subfolder of Delphi. To do this, go to the Directories/Conditional page of the package Project Options, and set the current directory (a single dot, for short) for the Output directory (for the BPL) and DCP output directory. Then compile the package and do not install it in Delphi—there's no need to.

At this point, you can create a normal application and write the standard code you'll use in a program to show a secondary form, as in the following listing:

```
uses
  PackScrollF;

procedure TForm1.BtnChangeClick(Sender: TObject);
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  try
    // initialize the data
    FormScroll.SelectedColor := Color;
    // show the form
    if FormScroll.ShowModal = mrOK then
      Color := FormScroll.SelectedColor;
  finally
    FormScroll.Free;
  end;
end;

procedure TForm1.BtnSelectClick(Sender: TObject);
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  // initialize the data and UI
  FormScroll.SelectedColor := Color;
  FormScroll.BitBtn1.Caption := 'Apply';
  FormScroll.BitBtn1.OnClick := FormScroll.ApplyClick;
  FormScroll.BitBtn2.Kind := bkClose;
  // show the form
  FormScroll.Show;
end;
```

One of the advantages of this approach is that you can refer to a form compiled into a package with the same code you'd use for a form compiled in the program. If you compile this program, the unit of the form will be bound to the program. To keep the form's unit in the package, you'll have to use run-time packages for the application and manually add the PackWithForm package to the list of run-time packages (this is not suggested by the Delphi IDE, because you have not installed the package in the development environment).

Once you've performed this step, compile the program; it will behave as usual. But now the form is in a DLL package, and you can modify the form in the package, recompile it, and run the application to see the effects. Notice, though, that for most changes affecting the `interface` portion of the package's units (for example, adding a component or a method to the form), you should also recompile the executable program calling the package.

NOTE *You can find the package and the program testing it in the* `PackForm` *folder of the source code related to the current chapter. The code for the next example is in the same folder. The package and projects are all referenced by the project group (BPG) file within the folder.*

Loading Packages at Run Time

In the previous example, I indicated that the PackWithForm package is a run-time package to be used by the application. This means the package is required to run the application and is loaded when the program starts, just as with the typical use of DLLs. You can avoid both aspects by loading the package dynamically, as you've done with DLLs. The resulting program will be more flexible, start more quickly, and use less memory.

An important element to keep in mind is that you'll need to call the `LoadPackage` and `UnloadPackage` Delphi functions rather than the `LoadLibrary`/`SafeLoadLibrary` and `FreeLibrary` Windows API functions. The functions provided by Delphi load the packages, but they also call their proper initialization and finalization code.

Besides this important element—which is easy to accomplish once you know about it—the program will require some extra code, because you cannot refer from the main program to the unit hosting the form. You cannot use the form class directly, nor access its properties or components—at least, not with the standard Delphi code. Both issues, however, can be solved using class references, class registration, and RTTI (run-time type information).

Let me begin with the first approach. In the form unit, in the package, I've added this initialization code:

```
initialization
  RegisterClass (TFormScroll);
```

As the package is loaded, the main program can use Delphi's `GetClass` function to get the class reference of the registered class and then call the `Create` constructor for this class reference.

To solve the second problem, I've made the `SelectedColor` property of the form in the package a published property, so that it is accessible via RTTI. Then I've replaced the code accessing this property (`FormScroll.Color`) with the following:

```
SetPropValue (FormScroll, 'SelectedColor', Color);
```

Summing up all these changes, here is the code used by the main program (the DynaPackForm application) to show the modal form from the dynamically loaded package:

```
procedure TForm1.BtnChangeClick(Sender: TObject);
var
  FormScroll: TForm;
  FormClass: TFormClass;
  HandlePack: HModule;
```

```
begin
  // try to load the package
  HandlePack := LoadPackage ('PackWithForm.bpl');
  if HandlePack > 0 then
  begin
    FormClass := TFormClass(GetClass ('TFormScroll'));
    if Assigned (FormClass) then
    begin
      FormScroll := FormClass.Create (Application);
      try
        // initialize the data
        SetPropValue (FormScroll, 'SelectedColor', Color);
        // show the form
        if FormScroll.ShowModal = mrOK then
          Color := GetPropValue (FormScroll, 'SelectedColor');
      finally
        FormScroll.Free;
      end;
    end
    else
      ShowMessage ('Form class not found');
    UnloadPackage (HandlePack);
  end
  else
    ShowMessage ('Package not found');
end;
```

Notice that the program unloads the package as soon as it is done with it. This step is not compulsory. I could have moved the `UnloadPackage` call in the `OnDestroy` handler of the form, and avoided reloading the package after the first time.

Now you can try running this program without the package available. You'll see that it starts properly, only to complain that it cannot find the package as you click the Change button. In this program, you don't need to use run-time packages to keep the unit outside your executable file, because you are not referring to the unit in your code. Also, the PackWithForm package doesn't need to be listed in the run-time packages. However, you must use run-time packages for it to work at all, or else your program will include VCL global variables (such as the `Application` object) and the dynamically loaded package will include another version, because it will refer to the VCL packages anyway.

WARNING *When a program that loads a package dynamically is closed, you may experience access violations. Frequently, they occur because an object whose class is defined in the package is kept in memory even after the package is unloaded. When the program shuts down, it may try to free that object by calling the* `Destroy` *method of a non-existing VMT, and thus cause the error. Having said this, I know by experience that these types of errors are very difficult to track and fix. I suggest that you make sure to destroy all the objects before unloading the package.*

Using Interfaces in Packages

Accessing forms' classes by means of methods and properties is much simpler than using RTTI all over the place. To build a larger application, I definitely try to use interfaces and to have multiple forms, each implementing a few standard interfaces defined by the program. An example cannot really do justice to this type of architecture, which becomes relevant for a large program, but I've tried to build a program to show how this idea can be applied in practice.

NOTE *If you don't know much about interfaces, I suggest you refer to the related portion of Chapter 2, "The Delphi Programming Language," before reading this section.*

To build the IntfPack project, I've used three packages plus a demo application. Two of the three packages (IntfFormPack and IntfFormPack2) define alternative forms used to select a color. The third package (IntfPack) hosts a shared unit, used by both other packages. This unit includes the definition of the interface. I couldn't add it to both other packages because you cannot load two packages that have the same name with a unit (even by run-time loading).

The IntfPack package's only file is the IntfColSel unit, displayed in Listing 10.1. This unit defines the common interface (you'll probably have a number of them in real-world applications) plus a list of registered classes; it mimics Delphi's `RegisterClass` approach, but makes available the complete list so that you can easily scan it.

LISTING 10.1: THE INTFCOLSEL UNIT OF THE INTFPACK PACKAGE

```
unit IntfColSel;

interface

uses
  Graphics, Contnrs;

type
  IColorSelect = interface
  ['{3F961395-71F6-4822-BD02-3B475FF516D4}']
    function Display (Modal: Boolean = True): Boolean;
    procedure SetSelColor (Col: TColor);
    function GetSelColor: TColor;
    property SelColor: TColor
      read GetSelColor write SetSelColor;
  end;

procedure RegisterColorSelect (AClass: TClass);

var
  ClassesColorSelect: TClassList;

implementation

procedure RegisterColorSelect (AClass: TClass);
begin
  if ClassesColorSelect.IndexOf (AClass) < 0 then
    ClassesColorSelect.Add (AClass);
end;
```

```
initialization
  ClassesColorSelect := TClassList.Create;

finalization
  ClassesColorSelect.Free;

end.
```

Once you have this interface available, you can define forms that implement it, as in the following example taken from IntfFormPack:

```
type
  TFormSimpleColor = class(TForm, IColorSelect)
    ...
  private
    procedure SetSelColor (Col: TColor);
    function GetSelColor: TColor;
  public
    function Display (Modal: Boolean = True): Boolean;
```

The two access methods read and write the value of the color from some components of the form (a ColorGrid control in this case), whereas the `Display` method internally calls either `Show` or `ShowModal`, depending on the parameter:

```
function TFormSimpleColor.Display(Modal: Boolean): Boolean;
begin
  Result := True; // default
  if Modal then
    Result := (ShowModal = mrOK)
  else
  begin
    BitBtn1.Caption := 'Apply';
    BitBtn1.OnClick := ApplyClick;
    BitBtn2.Kind := bkClose;
    Show;
  end;
end;
```

As you can see from this code, when the form is modeless the OK button is turned into an Apply button. Finally, the unit has the registration code in the `initialization` section, so that it is executed when the package is dynamically loaded:

```
RegisterColorSelect (TFormSimpleColor);
```

The second package, IntfFormPack2, has a similar architecture but a different form. You can look it up in the source code (I've not discussed the second form here as its code doesn't add much to the structure of the example).

With this architecture in place, you can build a rather elegant and flexible main program, which is based on a single form. When the form is created, it defines a list of packages (HandlesPackages) and loads them all. I've hard-coded the package in the code of the example, but of course you can search for the packages of the current folder or use a configuration file to make the application structure more flexible. After loading the packages, the program shows the registered classes in a list box. This is the code of the `LoadDynaPackage` and `FormCreate` methods:

```
procedure TFormUseIntf.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  // loads all runtime packages
  HandlesPackages := TList.Create;
  LoadDynaPackage ('IntfFormPack.bpl');
  LoadDynaPackage ('IntfFormPack2.bpl');

  // add class names and select the first
  for I := 0 to ClassesColorSelect.Count - 1 do
    lbClasses.Items.Add (ClassesColorSelect [I].ClassName);
  lbClasses.ItemIndex := 0;
end;

procedure TFormUseIntf.LoadDynaPackage(PackageName: string);
var
  Handle: HModule;
begin
  // try to load the package
  Handle := LoadPackage (PackageName);
  if Handle > 0 then
    // add to the list for later removal
    HandlesPackages.Add (Pointer(Handle))
  else
    ShowMessage ('Package ' + PackageName + ' not found');
end;
```

The main reason for keeping the list of package handles is to be able to unload them all when the program ends. You don't need these handles to access the forms defined in those packages; the run-time code used to create and show a form uses the corresponding component classes. This is a snippet of code used to display a modeless form (an option controlled by a check box):

```
var
  AComponent: TComponent;
  ColorSelect: IColorSelect;
begin
  AComponent := TComponentClass
    (ClassesColorSelect[LbClasses.ItemIndex]).Create (Application);
  ColorSelect := AComponent as IColorSelect;
  ColorSelect.SelColor := Color;
  ColorSelect.Display (False);
```

The program uses the `Supports` function to check that the form really does support the interface before using it, and also accounts for the modal version of the form; but its essence is properly depicted in the preceding four statements.

By the way, notice that the code doesn't require a form. A nice exercise would be to add to the architecture a package with a component encapsulating the color selection dialog box or inheriting from it.

WARNING *The main program refers to the unit hosting the interface definition but should not link this file in. Rather, it should use the run-time package containing this unit, as the dynamically loaded packages do. Otherwise the main program will use a different copy of the same code, including a different list of global classes. It is this list of global classes that should not be duplicated in memory.*

The Structure of a Package

You may wonder whether it is possible to know if a unit has been linked in the executable file or if it's part of a run-time package. Not only is this possible in Delphi, but you can also explore the overall structure of an application. A component can use the undocumented `ModuleIsPackage` global variable, declared in the SysInit unit. You should never need this variable, but it is technically possible for a component to have different code depending on whether it is packaged. The following code extracts the name of the run-time package hosting the component, if any:

```
var
  fPackName: string;
begin
  // get package name
  SetLength (fPackName, 100);
  if ModuleIsPackage then
  begin
    GetModuleFileName (HInstance, PChar (fPackName), Length (fPackName));
    fPackName := PChar (fPackName) // string length fixup
  end
  else
    fPackName := 'Not packaged';
```

Besides accessing package information from within a component (as in the previous code), you can also do so from a special entry point of the package libraries, the `GetPackageInfoTable` function. This function returns some specific package information that Delphi stores as resources and includes in the package DLL. Fortunately, you don't need to use low-level techniques to access this information, because Delphi provides some high-level functions to manipulate it.

You can use two functions to access package information:

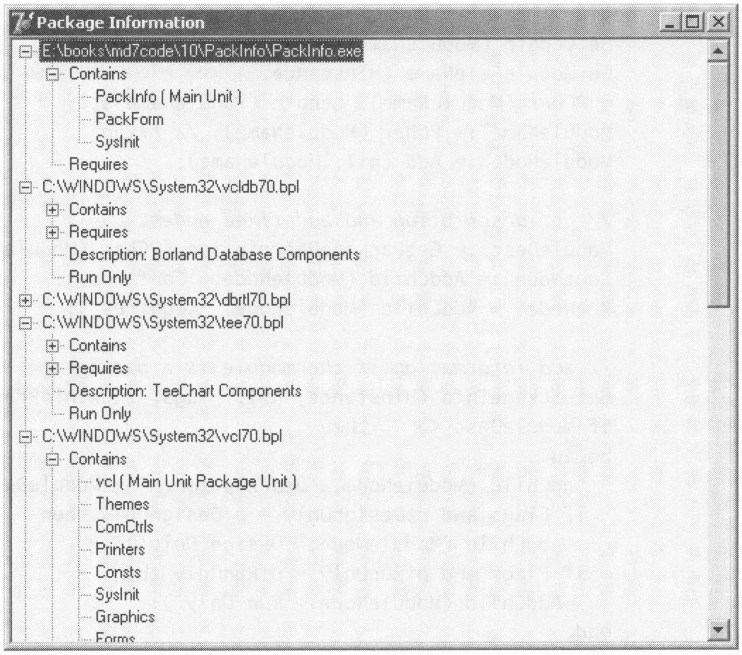
- `GetPackageDescription` returns a string that contains a description of the package. To call this function, you must supply the name of the module (the package library) as the only parameter.
- `GetPackageInfo` doesn't directly return information about the package. Instead, you pass it a function that it calls for every entry in the package's internal data structure. In practice, `GetPackageInfo` will call your function for every one of the package's contained units and required packages. In addition, `GetPackageInfo` sets several flags in an Integer variable.

These two function calls allow you to access internal information about a package, but how do you know which packages your application is using? You could determine this information by looking at an executable file using low-level functions, but Delphi helps you again by supplying a simpler approach. The `EnumModules` function doesn't directly return information about an application's modules; but it lets you pass it a function, which it calls for each module of the application, for the main executable file, and for each of the packages the application relies on.

To demonstrate this approach, I've built a program that displays the module and package information in a TreeView component. Each first-level node corresponds to a module; within each module I've built a subtree that displays the contained and required packages for that module, as well as the package description and compiler flags (`RunOnly` and `DesignOnly`). You can see the output of this example in Figure 10.6.

FIGURE 10.6

The output of the PackInfo example, with details of the packages it uses



```
E:\books\md7code\10\PackInfo\PackInfo.exe
  Contains
    PackInfo ( Main Unit )
    PackForm
    SysInit
  Requires
C:\WINDOWS\System32\vcldb70.bpl
  Contains
  Requires
  Description: Borland Database Components
  Run Only
C:\WINDOWS\System32\dbrtl70.bpl
C:\WINDOWS\System32\tee70.bpl
  Contains
  Requires
  Description: TeeChart Components
  Run Only
C:\WINDOWS\System32\vcl70.bpl
  Contains
    vcl ( Main Unit Package Unit )
    Themes
    ComCtrls
    Printers
    Consts
    SysInit
    Graphics
    Forms
```

In addition to the TreeView component, I've added several other components to the main form but hidden them from view: a DBEdit, a Chart, and a FilterComboBox. I added these components simply to include more run-time packages in the application, beyond the ubiquitous Vcl and Rtl packages. The only method of the form class is `FormCreate`, which calls the module enumeration function:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  EnumModules(ForEachModule, nil);
end;
```

The `EnumModules` function accepts two parameters: the callback function (in this case, `ForEachModule`) and a pointer to a data structure that the callback function will use (in this case, `nil`, because you don't need this). The callback function must accept two parameters—an `HInstance` value and an untyped pointer—and must return a Boolean value. The `EnumModules` function will, in turn, call your callback function for each module, passing the instance handle of each module as the first parameter and the data structure pointer (`nil` in this example) as the second:

```
function ForEachModule (HInstance: Longint;
  Data: Pointer): Boolean;
var
  Flags: Integer;
  ModuleName, ModuleDesc: string;
  ModuleNode: TTreeNode;
begin
  with Form1.TreeView1.Items do
  begin
    SetLength (ModuleName, 200);
    GetModuleFileName (HInstance,
      PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar (ModuleName); // fixup
    ModuleNode := Add (nil, ModuleName);

    // get description and add fixed nodes
    ModuleDesc := GetPackageDescription (PChar (ModuleName));
    ContNode := AddChild (ModuleNode, 'Contains');
    ReqNode := AddChild (ModuleNode, 'Requires');

    // add information if the module is a package
    GetPackageInfo (HInstance, nil, Flags, ShowInfoProc);
    if ModuleDesc <> '' then
    begin
      AddChild (ModuleNode, 'Description: ' + ModuleDesc);
      if Flags and pfDesignOnly = pfDesignOnly then
        AddChild (ModuleNode, 'Design Only');
      if Flags and pfRunOnly = pfRunOnly then
        AddChild (ModuleNode, 'Run Only');
    end;
  end;
  Result := True;
end;
```

As you can see in the preceding code, the `ForEachModule` function begins by adding the module name as the main node of the tree (by calling the `Add` method of the `TreeView1.Items` object and passing `nil` as the first parameter). It then adds two fixed child nodes, which are stored in the `ContNode` and `ReqNode` variables declared in the `implementation` section of this unit.

Next, the program calls the `GetPackageInfo` function and passes it another callback function, `ShowInfoProc`, which I'll discuss shortly, to provide a list of the application's or package's units. At the end of the `ForEachModule` function, if the module is a package the program adds more information, such as its description and compiler flags (the program knows it's a package if its description isn't an empty string).

Earlier, I mentioned passing another callback function (the `ShowInfoProc` procedure) to the `GetPackageInfo` function, which in turn calls the callback function for each contained or required package of a module. This procedure creates a string that describes the package and its main flags (added within parentheses), and then inserts that string under one of the two nodes (`ContNode` and `ReqNode`), depending on the type of the module. You can determine the module type by examining the `NameType` parameter. Here is the complete code for the second callback function:

```
procedure ShowInfoProc (const Name: string; NameType: TNameType; Flags: Byte;
  Param: Pointer);
var
  FlagStr: string;
begin
  FlagStr := ' ';
  if Flags and ufMainUnit <> 0 then
    FlagStr := FlagStr + 'Main Unit ';
  if Flags and ufPackageUnit <> 0 then
    FlagStr := FlagStr + 'Package Unit ';
  if Flags and ufWeakUnit <> 0 then
    FlagStr := FlagStr + 'Weak Unit ';
  if FlagStr <> ' ' then
    FlagStr := ' (' + FlagStr + ')';
  with Form1.TreeView1.Items do
    case NameType of
      ntContainsUnit: AddChild (ContNode, Name + FlagStr);
      ntRequiresPackage: AddChild (ReqNode, Name);
    end;
end;
```

What's Next?

In this chapter, you have seen how you can call functions that reside in DLLs and how to create DLLs using Delphi. After discussing dynamic libraries in general, I focused on Delphi packages, covering in particular how to place forms and other classes in a package. This is a handy technique for dividing a Delphi application into multiple executable files. While discussing packages, I explained how advanced techniques including RTTI and interfaces can be used to obtain dynamic and flexible application architectures.

I'll return to the topic of libraries that expose objects and classes when I discuss COM and OLE in Chapter 12, "From COM to COM+." For the moment, let's move to another topic related to the architecture of Delphi applications: the use of modeling tools and more examples of OOP-related techniques.

Chapter 11

Modeling and OOP Programming (with ModelMaker)

WHEN BORLAND DECIDED TO provide a UML design solution for the Enterprise and Architect editions of Delphi 7, it chose to bundle ModelMaker, by ModelMaker Tools of Holland (`www.modelmakertools.com`). ModelMaker is a high-quality UML design tool with integration into the Delphi IDE. But as you become acquainted with ModelMaker and over the course of this chapter, you'll see that ModelMaker is far more than a UML diagramming tool. I will, of course, cover the UML diagramming capabilities of ModelMaker, but I will also spend some time on the other features of the tool as well a conceptual overview of the product that should allow you to begin getting the most out of it.

ModelMaker has been around since the early days of Delphi, and over time it has accumulated options to support almost the entire Delphi language as well as a vast number of conveniences for programmers. The result is a huge feature set that can be daunting at first glance. The ModelMaker user interface comprises more than 100 forms, and without the proper grounding the uninitiated may become frustrated. Stick with me, and you will soon be navigating ModelMaker fearlessly.

Although ModelMaker is often referred to as a UML diagramming tool, I prefer to describe it as a Delphi-specific, customizable, extensible full-cycle UML diagramming and CASE tool. It's Delphi-specific because it is designed to handle Delphi code. When working with a property, for instance, the dialog boxes in ModelMaker present options that are specific to Delphi language keywords and concepts. ModelMaker is customizable because, as you'll see, hundreds of options control how Delphi code is generated from your object model. ModelMaker is extensible because it includes a robust OpenTools API that allows the creation of plug-in experts to extend the functionality of the product. It's is a full-cycle tool because it offers features that apply to all phases of the standard development cycle. Finally, ModelMaker can be described as a CASE tool because it will automatically generate some of the redundant, obvious code required for Delphi classes, leaving it to you to provide the operational code for your classes.

NOTE *This chapter has been co-written with Robert Leahey and benefits from his in-depth knowledge of and extensive experience with ModelMaker. In the world of software, Robert is an architect, programmer, author, and speaker. As a musician, he has played professionally for over 20 years and is currently a graduate student at the University of North Texas in the area of music theory. Via his company, Thoughtsmithy (*`www.thoughtsmithy.com`*), Robert offers consulting and training services, commercial software, audio production, and large-scale LEGO brick sculptures. He lives in north Texas with his wife and daughters.*

This chapter covers the following:

- Understanding ModelMaker
- Modeling and UML
- Coding features of ModelMaker
- Documentation and macros
- Refactoring your code
- Implementing patterns

Understanding ModelMaker's Internal Model

Before proceeding with a discussion of ModelMaker's UML support and other features, it is vital that you understand conceptually how ModelMaker manages your code model. Unlike Delphi and other editors, ModelMaker does not continually parse a source code file to visually represent the contents. Consider Delphi: Any IDE convenience you use to alter your code will directly change the contents of the source code file (which you can then save in order to persist the changes). In contrast, ModelMaker maintains an internal model representing your classes, code, documentation, and so on, from which your source code files are generated. When you edit your model through the various editors in ModelMaker, the changes are applied to the internal model—not the external source code files; at least, not until you tell ModelMaker to regenerate the external files. Understanding this distinction should save you some frustration.

Another concept to understand is that ModelMaker is capable of representing a single internal code model with multiple views in its user interface. The model can be viewed and edited, for example, as a class hierarchy, or as a list of units with contained classes. Class members can be sorted, filtered, grouped, and edited in a variety of ways. Any number of views can be seen in the various plug-ins available for ModelMaker. But most important for this discussion, the UML diagram editor itself is another view into the model. When you visualize elements of the model (such as classes and units) in your diagrams, you are creating visual representations of the code model elements; if you delete a symbol from a diagram, you are not necessarily deleting the element from the model—you are simply removing the representation from your diagram.

One last consideration about diagramming in ModelMaker: Although ModelMaker offers several wizards and automation features in the area of visualization, the product will not read your code and magically produce attractive UML diagrams with no effort on your part. Upon importing your source code and adding your classes to diagrams, you will need to arrange the symbols in order to create usable UML diagrams.

Modeling and UML

UML (Unified Modeling Language) is a graphical notation used to express the analysis and design of a software project and communicate it to others. UML is language independent, but it's intended to describe object-oriented projects. As the creators of UML stress, it is not itself a methodology; it can be used as a descriptive tool no matter what your preferred design process.

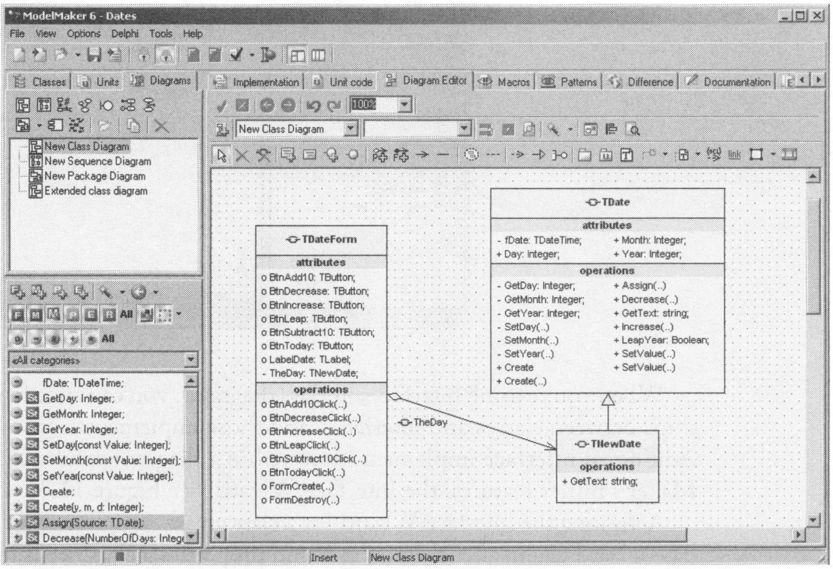
My goal is to look at UML diagrams from the perspective of a Delphi programmer using ModelMaker. An in-depth discussion of UML is well beyond the scope of this chapter.

NOTE *The best introduction to UML I've seen is Martin Fowler's compact* UML Distilled *(Addison-Wesley, 1999).*

Class Diagrams

One of the most common UML diagrams supported by ModelMaker is the class diagram. Class diagrams can display a wide variety of class relationships, but at its simplest this type of diagram depicts a set of classes or objects and the static relationships between them. For example, Figure 11.1 is a class diagram containing the classes from the NewDate program presented in Chapter 2, "The Delphi Programming Language." If the results are different when you import these classes into ModelMaker and create your own class diagram, keep in mind the numerous options I discussed earlier. Many settings control how your visualized classes will appear. You can open the ModelMaker file (MPB file) used for Figure 11.1 from the corresponding source code folder of the current chapter.

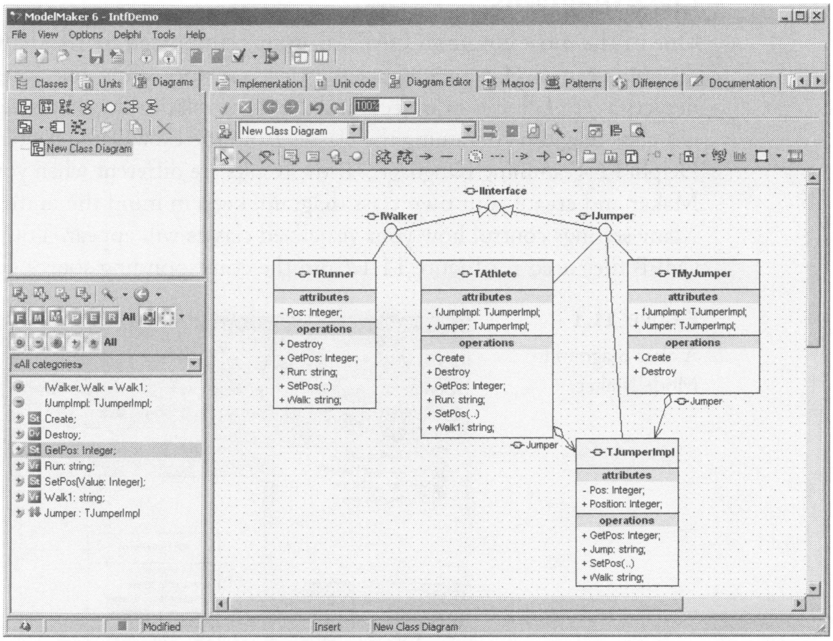
FIGURE 11.1
A class diagram in ModelMaker


ModelMaker 6 - Dates
File View Options Delphi Tools Help
Classes Units Diagrams
Implementation Unit code Diagram Editor Macros Patterns Difference Documentation
100%
New Class Diagram
New Class Diagram
New Sequence Diagram
New Package Diagram
Extended class diagram
All
All
«All categories»
fDate: TDateTime;
GetDay: Integer;
GetMonth: Integer;
GetYear: Integer;
SetDay(const Value: Integer);
SetMonth(const Value: Integer);
SetYear(const Value: Integer);
Create;
Create(y, m, d: Integer);
Assign(Source: TDate);
Decrease(NumberOfDays: Intege
TDateForm
attributes
o BtnAdd10: TButton;
o BtnDecrease: TButton;
o BtnIncrease: TButton;
o BtnLeap: TButton;
o BtnSubtract10: TButton;
o BtnToday: TButton;
o LabelDate: TLabel;
- TheDay: TNewDate;
operations
o BtnAdd10Click(..)
o BtnDecreaseClick(..)
o BtnIncreaseClick(..)
o BtnLeapClick(..)
o BtnSubtract10Click(..)
o BtnTodayClick(..)
o FormCreate(..)
o FormDestroy(..)
TheDay
TDate
attributes
- fDate: TDateTime;
+ Month: Integer;
+ Day: Integer;
+ Year: Integer;
operations
- GetDay: Integer;
+ Assign(..)
- GetMonth: Integer;
+ Decrease(..)
- GetYear: Integer;
+ GetText: string;
- SetDay(..)
+ Increase(..)
- SetMonth(..)
+ LeapYear: Boolean;
- SetYear(..)
+ SetValue(..)
+ Create
+ SetValue(..)
+ Create(..)
TNewDate
operations
+ GetText: string;
Insert
New Class Diagram

Earlier, I mentioned that the ModelMaker diagram editor is just another view into the internal model. Some of the symbols in the ModelMaker diagrams map directly to code model elements, and others do not. With low-level diagrams like class diagrams, most symbols represent actual code model elements. Manipulating these symbols can change the code generated by ModelMaker. At the opposite end of the spectrum, in use case diagrams most (if not all) symbols have no representation within the code model. In your class diagram, you can add new classes, interfaces, fields, properties, and even documentation to the model. Likewise, you can change the inheritance of a class in the model from within the diagram. At the same time, you can add several symbols to a class diagram that have no logical representation within the code model.

Class diagrams in ModelMaker also allow you to code to interfaces, thus working at a higher abstraction level. Figure 11.2 shows the relationships of the classes and interfaces in a complex example of interface use, IntfDemo. This example is covered in an online book discussed in Appendix C, "Free Companion Books on Delphi," and is available among the chapter's source code examples.

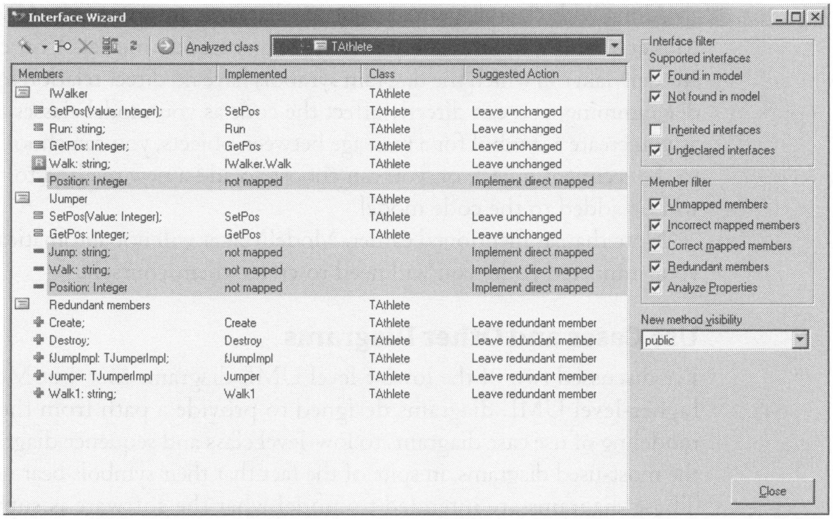
FIGURE 11.2
A class diagram with interfaces, classes, and interface delegation


ModelMaker 6 - IntfDemo
File View Options Delphi Tools Help
Classes Units Diagrams
Implementation Unit code Diagram Editor Macros Patterns Difference Documentation
100%
New Class Diagram
New Class Diagram
All
All
«All categories»
IWalker.Walk = Walk1;
fJumpImpl: TJumperImpl;
Create;
Destroy;
GetPos: Integer;
Run: string;
SetPos(Value: Integer);
Walk1: string;
Jumper : TJumperImpl
IInterface
IWalker
IJumper
TRunner
attributes
- Pos: Integer;
operations
+ Destroy
+ GetPos: Integer;
+ Run: string;
+ SetPos(..)
+ Walk: string;
TAthlete
attributes
- fJumpImpl: TJumperImpl;
+ Jumper: TJumperImpl;
operations
+ Create
+ Destroy
+ GetPos: Integer;
+ Run: string;
+ SetPos(..)
+ Walk1: string;
TMyJumper
attributes
- fJumpImpl: TJumperImpl;
+ Jumper: TJumperImpl;
operations
+ Create
+ Destroy
Jumper
Jumper
TJumperImpl
attributes
- Pos: Integer;
+ Position: Integer;
operations
+ GetPos: Integer;
+ Jump: string;
+ SetPos(..)
+ Walk: string;
Modified
Insert
New Class Diagram

When you're using interfaces in class diagrams, you can specify interface implementation relationships between classes and interfaces, and those implementations will be added to the code model. Adding an interface implementation within a diagram results in the appearance of one of ModelMaker's niftier features: the Interface Wizard (see Figure 11.3).

Activating the Interface Wizard for a class greatly simplifies the task of implementing an interface. The wizard enumerates the methods and properties a class needs in order to implement a given interface (or interfaces); if told to, the wizard will add those needed members to the implementing class. Note that it is up to you to provide meaningful code for any methods added to the class. In Figure 11.3, the wizard is evaluating `TAthlete` for its implementation of `IWalker` and `IJumper` and suggesting the changes that are necessary for correct implementation of these interfaces.

FIGURE 11.3
ModelMaker's Interface Wizard



Interface Wizard
Analyzed class
Close

Analyzed class: TAthlete

Members	Implemented	Class	Suggested Action
IWalker		TAthlete	
SetPos(Value: Integer);	SetPos	TAthlete	Leave unchanged
Run: string;	Run	TAthlete	Leave unchanged
GetPos: Integer;	GetPos	TAthlete	Leave unchanged
Walk: string;	IWalker.Walk	TAthlete	Leave unchanged
Position: Integer	not mapped		Implement direct mapped
IJumper		TAthlete	
SetPos(Value: Integer);	SetPos	TAthlete	Leave unchanged
GetPos: Integer;	GetPos	TAthlete	Leave unchanged
Jump: string;	not mapped		Implement direct mapped
Walk: string;	not mapped		Implement direct mapped
Position: Integer	not mapped		Implement direct mapped
Redundant members		TAthlete	
Create;	Create	TAthlete	Leave redundant member
Destroy;	Destroy	TAthlete	Leave redundant member
fJumpImpl: TJumperImpl;	fJumpImpl	TAthlete	Leave redundant member
Jumper: TJumperImpl	Jumper	TAthlete	Leave redundant member
Walk1: string;	Walk1	TAthlete	Leave redundant member

Interface filter
Supported interfaces
☑ Found in model
☑ Not found in model
☐ Inherited interfaces
☑ Undeclared interfaces

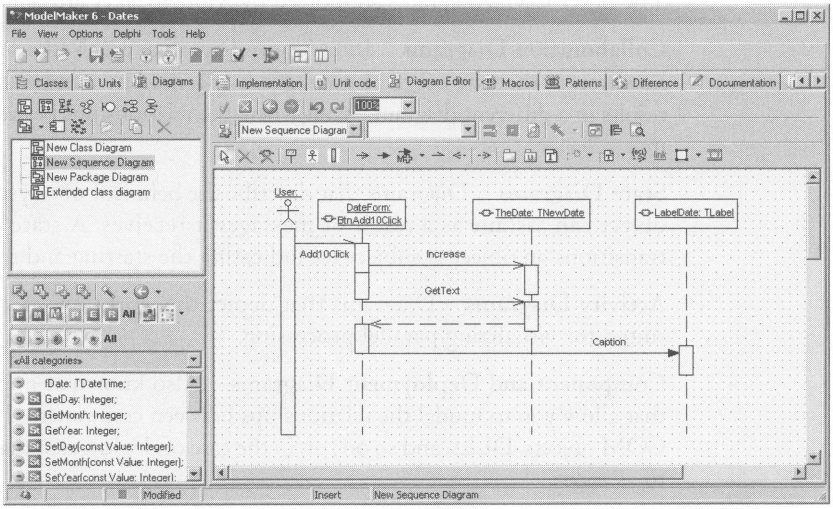
Member filter
☑ Unmapped members
☑ Incorrect mapped members
☑ Correct mapped members
☑ Redundant members
☑ Analyze Properties

New method visibility: public

Sequence Diagrams

Sequence diagrams model object interaction by rendering objects and the messages that pass between them over time. In a typical sequence diagram, the objects interacting within a system are arrayed along the x-axis, and time is represented as passing from top to bottom along the y-axis. Messages are represented as arrows between objects. You can see an example of a rather trivial sequence diagram in Figure 11.4. Sequence diagrams can be created at various levels of abstraction, allowing you to represent high-level system interaction involving just a few messages or low-level interaction with many messages.

FIGURE 11.4
A sequence diagram for an event handler of the NewDate example



ModelMaker 6 - Dates
File View Options Delphi Tools Help
Classes
Units
Diagrams
Implementation
Unit code
Diagram Editor
Macros
Patterns
Difference
Documentation
New Class Diagram
New Sequence Diagram
New Package Diagram
Extended class diagram
New Sequence Diagram
100%
All
All
«All categories»
fDate: TDateTime;
GetDay: Integer;
GetMonth: Integer;
GetYear: Integer;
SetDay(const Value: Integer);
SetMonth(const Value: Integer);
SetYear(const Value: Integer);
User:
DateForm: BtnAdd10Click
TheDate: TNewDate
LabelDate: TLabel
Add10Click
Increase
GetText
Caption
Modified
Insert
New Sequence Diagram

Along with class diagrams, sequence diagrams are among the UML diagrams supported by ModelMaker that are most closely related to your code model. You can create several diagrams in ModelMaker in which the diagram symbols have no direct relation to your code model, but in sequence diagramming, you can directly affect the code as you model classes and their methods. For example, as you create a symbol for a message between objects, you can choose from a list of methods belonging to the recipient object; or, you can choose to add a new method to the object, and that new method will be added to the code model.

Note that, as mentioned earlier, ModelMaker will not automatically create sequence diagrams from your imported code; you will need to create them yourself.

Use Cases and Other Diagrams

I've discussed two of the lowest-level UML diagrams first, but ModelMaker supports several other higher-level UML diagrams designed to provide a path from the highest-level user interaction modeling of use case diagrams to low-level class and sequence diagrams. Use case diagrams are among the most-used diagrams, in spite of the fact that their symbols bear no relation to code model elements. These diagrams are intended to model what the software is supposed to do, and they are self-explanatory enough to use in analysis sessions with non-developers.

A simple use case diagram consists of *actors* (users or application subsystems) and *use cases* (things the actors do). One of the most frequent questions regarding use cases is how to handle use case texts in ModelMaker. *Texts* for use cases are a typical next step when doing preliminary analysis. For example, a use case is a short description of an action an actor might take ("Preview Sales Report" or "Resize Window"); a use case text is a longer, more detailed description of the text. ModelMaker does not specifically support the longer use case texts; you can either use an annotation symbol within the diagram attached to the use case symbol, or you can link the use case symbol to an external file containing the use case text. You'll learn more about these techniques in the "Common Diagram Elements" section of this chapter.

The other UML diagrams supported by ModelMaker are as follows:

Collaboration Diagrams Interaction diagrams, much like sequence diagrams. They differ, however, in that the order of messages is specified by numbering rather than by time-scale. This results in a different diagram layout where the relationships between objects can sometimes be seen more clearly.

State Diagrams Diagrams that describe the behavior of a system by identifying all the states an object can assume as a result of messages it receives. A state diagram should list all the state transitions an object is subject to, indicating the starting and resulting state of each transition.

Activity Diagrams Diagrams that depict the workflow of a system and are particularly well suited for visualizing parallel processing.

Component and Deployment Diagrams Also known as implementation diagrams. Diagrams that allow you to model the relationships between components (modules, actually—executables, COM objects, DLLs, and so on) or, in the case of deployment diagrams, physical resources (referred to as *nodes*).

Non-UML Diagrams

ModelMaker supports three diagrams that are not UML-standard, but are quite useful:

Mind-Map Diagrams Originated by Tony Buzan in the 1960s. An excellent method for brainstorming, exploring branching topics, or quickly recording related thoughts. I've often used mind-map diagrams for generic data display during presentations.

Unit Dependency Diagrams Often used to display the results of ModelMaker's powerful Unit Dependency Analyzer. These diagrams can show the branching relations of units within a Delphi project.

Robustness Diagrams Questionably left out of the UML specification. These diagrams help to bridge the gap between interface-only use case modeling and implementation-specific sequence diagrams. Robustness diagrams can help an analysis team verify their use case and begin looking toward implementation details.

Common Diagram Elements

Each type of diagram supported by ModelMaker contains symbols specific to that diagram type, but there are elements within the Diagram Editor that are common to all diagram types. You can add images and shapes to your diagrams, as well as package symbols (containers to which you can add other symbols).

The Hyperlink symbol lets you add to a diagram a label linked to some other entity. In fact, the vast majority of diagram symbols support this hyperlinking feature. You can link to another diagram (clicking the link will open the linked diagram in the editor), you can link to an element within the code model (be it class, unit, method, interface, and so on—clicking this link will open the appropriate editor for the linked element), or you can link to an external document (this link will open the linked document with the appropriate application).

Three different types of annotation tools are available to each diagram type. Several documents more fully explain these tools, so suffice it to say here that you can add a stand-alone annotation symbol; you can add an annotation symbol that automatically displays the linked object's internal documentation; or you can add an annotation symbol, type in your text, and link this symbol to an object. When you do this, the object's internal documentation is updated to match the annotation symbol's text.

Relationship or association symbols default to straight lines. However, you can turn them into orthogonal lines by selecting the symbol and pressing Ctrl+O. You can also add nodes to these symbols by pressing Ctrl and clicking the line. ModelMaker attempts to keep these symbols orthogonal when at all possible.

ModelMaker also offers a robust set of visual symbol styles. You can define font and color styles in a hierarchical manner and apply them by name to your diagram symbols. See the entry "style manager" in the online help for more information.

One final common feature to note is the ability to hierarchically order the Diagrams list (see Figure 11.5). You can add folders for organizational purposes and to "re-parent" diagrams; to do so, Ctrl+drag a diagram to its new parent.

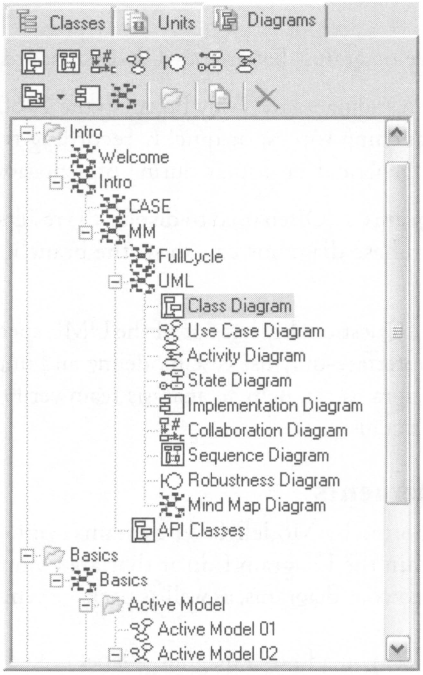

Classes
Units
Diagrams
Intro
Welcome
Intro
CASE
MM
FullCycle
UML
Class Diagram
Use Case Diagram
Activity Diagram
State Diagram
Implementation Diagram
Collaboration Diagram
Sequence Diagram
Robustness Diagram
Mind Map Diagram
API Classes
Basics
Basics
Active Model
Active Model 01
Active Model 02

FIGURE 11.5
The organizational possibilities of the Diagrams view

ModelMaker's diagramming capabilities encompass a vast set of possibilities; once you've put some time into learning the feature set, you may find that it transforms your development process. For the Delphi developer, the two-way active nature of the Diagram Editor provides a far more dynamic diagramming experience than most UML editors that simply generate static "pretty pictures."

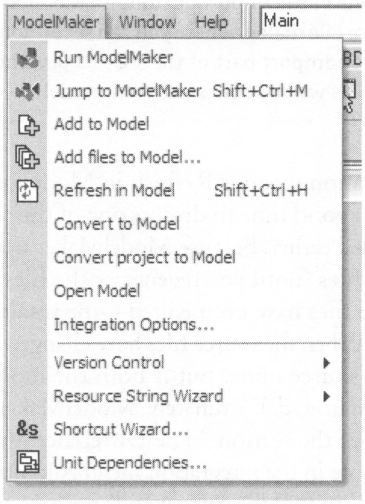
Coding Features of ModelMaker

As you've seen, ModelMaker is a powerful UML diagramming tool, and you can do a great deal of analysis and design work exercising only those capabilities; but, as I'll explain in the following sections, ModelMaker offers a great deal more than just diagramming. Many developers use ModelMaker as their primary development environment, supplanting Delphi in that regard. This is due in part to the visual way in which ModelMaker represents Delphi programming tasks, automating many of the repetitive parts of coding Delphi classes. But it is also because Delphi, for all its strengths, tends to facilitate the development of code where the line between the presentation domain and problem domain is blurred. In other words, it is easy to write application implementation code—the code that actually does stuff—and put it right in the event handlers of your forms. This is typically not considered good object-oriented design. On the other hand, ModelMaker facilitates the creation and refactoring of problem domain objects, and the de-coupling of those objects from the user interface.

Before we get to these topics, though, I'll first discuss how Delphi and ModelMaker work together.

Delphi/ModelMaker Integration

When properly installed, ModelMaker adds a menu to the Delphi IDE, appropriately labeled ModelMaker:


ModelMaker
Window
Help
Main
Run ModelMaker
Jump to ModelMaker Shift+Ctrl+M
Add to Model
Add files to Model...
Refresh in Model Shift+Ctrl+H
Convert to Model
Convert project to Model
Open Model
Integration Options...
Version Control
Resource String Wizard
Shortcut Wizard...
Unit Dependencies...

If you don't see this menu, you need to install ModelMaker's DLL-based wizard in the Delphi registry, as covered in the sidebar "Installing New DLL Wizards" at the end of Chapter 1. From this menu, you can control ModelMaker somewhat and quickly import your code into ModelMaker projects.

Most of the menu options are available only if ModelMaker is running. Once you start ModelMaker (either from the Run ModelMaker menu item or in the normal fashion), the other items will become available. The integration menu contains a number of ways to add your code to a model. The Add to Model, Add Files to Model, Convert to Model, and Convert Project to Model items cause ModelMaker to import the specified units: The *Add* items import units into the currently loaded model in ModelMaker, and the *Convert* items create a new model and import the units into it.

Convert Project to Model is a great place to begin—make sure you back up your code, and then select this menu item while one of your projects is open in Delphi. The entire project will be imported into a new model in ModelMaker.

WHERE'S THE VCL? INHERITANCE AND IMPORTING CODE IN MODELMAKER

Upon examining your newly imported code in ModelMaker, you may notice that only the units that are part of the project were imported. ModelMaker won't automatically import units that are "used" by your project—doing so would create inordinately large models with many unnecessarily imported classes. However, many of ModelMaker's nicer inheritance-related features require that ancestor classes exist in the code model. (For instance, when properly set up, changes to ancestor classes will automatically propagate down to overriding descendants.)

Continued on next page

WHERE'S THE VCL? INHERITANCE AND IMPORTING CODE IN MODELMAKER *(continued)*

Fortunately, you have many options when importing code. Although you can import via the ModelMaker integration menu in Delphi (or by dragging a unit from Windows Explorer into ModelMaker), the more flexible way is to use one of the Import buttons on ModelMaker's main toolbar. Importing units this way opens the Import Source File dialog, where you can set options for how to import the code. Taking advantage of these options lets you import part of the VCL as placeholder classes so that you can leverage ModelMaker's inheritance tools without bloating your model.

Also in the integration menu is Refresh in Model, which forces ModelMaker to re-import the current unit. This is a good time to discuss one of the consequences of ModelMaker's internal code model that I mentioned earlier. Because ModelMaker operates on its internal model and not on your external source code files (until you regenerate the files), it is common to find that both your model and your source code files have been edited—the result being that your model is now out of synch with the source files. When the source files have changed but not the model, you can re-synch the model by re-importing the source units. But if both the model and source files have been changed, the situation is more complicated. Fortunately, ModelMaker offers a robust set of tools to handle synchronization problems. See the section "The Difference View" for more information.

Another item of note in the integration menu is Jump to ModelMaker. When you select this item, ModelMaker attempts to find the current code position within its loaded model, bringing ModelMaker to the front in the process.

Although ModelMaker can be controlled from Delphi, the integration *is* two-way. Similar to the ModelMaker menu in the Delphi IDE, a Delphi menu appears in ModelMaker. In that menu are commands that let you jump from the currently selected model element to its corresponding position in the source code file in Delphi, as well as commands that cause Delphi to perform a syntax check, a compile, or a build. Thus you can edit your code, generate it, and compile it, all from within ModelMaker.

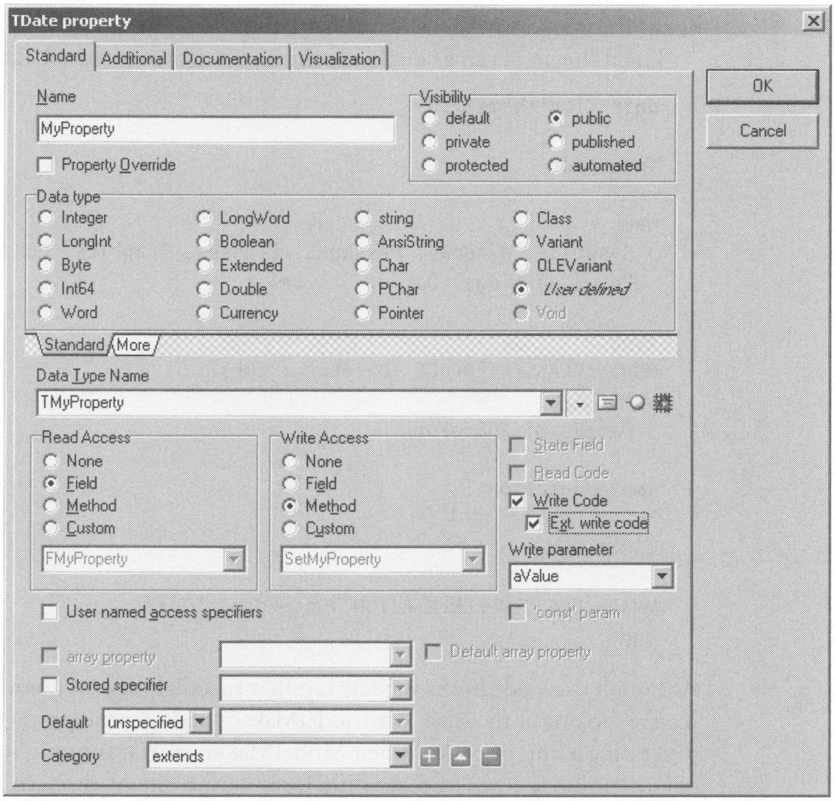
Managing the Code Model

It's time to discuss the nuts and bolts of coding within ModelMaker. Due to the objectified nature of ModelMaker's internal code model, editing code model elements is typically a more visual process than it is in Delphi. Editing a class property, for instance, is done via the Property Editor dialog, as you can see in Figure 11.6. Here is one of the best examples of ModelMaker's automation. When you add a new property, you don't have to worry about all the overhead of also adding a private state field, any read or write methods, or even the property's declaration. All you do is choose the appropriate settings in the editor, and ModelMaker creates the necessary supporting class members. This is more extended than the similar benefits offered in the Delphi IDE by Class Completion.

Notice that the attributes of a property that you might normally type in by hand are represented by various controls in the dialog. Visibility, Type, Read and Write specifications, and so on are all managed in the editor. The benefits are in the area of refactoring (not to mention the elimination of certain repetitive typing tasks). For instance, because ModelMaker manages a property as an object in its code model, changing something about the property—its type, for instance—will cause ModelMaker to apply that change to any references of which it is aware. If later you want to change the read access from a field to a method, you can make that change in the editor, and ModelMaker will take care of adding the get method and changing the property's declaration. Best of all, if you decide to rename the property or move it to another class, the property owns its supporting class members: They will be automatically renamed or moved as appropriate.

FIGURE 11.6

The Property Editor dialog



TDate property

Standard | Additional | Documentation | Visualization

Name: MyProperty

☐ Property Override

Visibility: ○ default ● public ○ private ○ published ○ protected ○ automated

Data type:
○ Integer ○ LongWord ○ string ○ Class
○ LongInt ○ Boolean ○ AnsiString ○ Variant
○ Byte ○ Extended ○ Char ○ OLEVariant
○ Int64 ○ Double ○ PChar ● *User defined*
○ Word ○ Currency ○ Pointer ○ Void

Standard | More

Data Type Name: TMyProperty

Read Access: ○ None ● Field ○ Method ○ Custom — FMyProperty

Write Access: ○ None ○ Field ● Method ○ Custom — SetMyProperty

☐ State Field
☐ Read Code
☑ Write Code
☑ Ext. write code

Write parameter: aValue

☐ 'const' param

☐ User named access specifiers

☐ array property

☐ Default array property

☐ Stored specifier

Default: unspecified

Category: extends

The same approach is used for each of the class member types; similar editors exist for methods, events, fields, and even method resolution clauses.

There's a developer-level sense of abstraction to developing in ModelMaker. It decouples you from the need to think about implementation details when editing class members; you need merely think in terms of interface, while ModelMaker handles most of the repetitive parts of implementing the member. (Don't confuse my metaphor with writing the code of a method implementation—you'll still need to do that.)

The Unit Code Editor

ModelMaker includes two code editors: the editor for implementing class methods, which I'll discuss next, and the Unit Code Editor, which requires some explanation. ModelMaker really is a class/object oriented tool—its conveniences are mostly built around managing class-level code. When it comes to code that is not part of a class (non-class type declarations, metaclass declarations, unit methods and variables, and so on), ModelMaker takes a more no-frills approach. When ModelMaker imports a unit, anything that can be stored within the code model is handled accordingly, and what's left over appears in the Unit Code Editor. (Often, for new users, this includes any documentation not residing within method implementations—but ModelMaker can reverse-engineer your documentation as well; more on that later.)

The following is an example of what you might see in the Unit Code Editor:

```
unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Dates, StdCtrls;

type
MMWIN:CLASSINTERFACE TDateForm; ID=37;
var
  DateForm: TDateForm;

implementation

{$R *.DFM}

MMWIN:CLASSIMPLEMENTATION TDateForm; ID=37;
end.
```

Although this code looks vaguely familiar to a Delphi programmer, it obviously will not compile. You're looking at the shell that ModelMaker's code generation engine will use when expanding or generating a unit of code. When ModelMaker generates a unit, it starts at the top of this code and begins emitting lines of text while looking for one of three things: plain text, macros, or code-generation tags.

Plain text, in this example, can be found in the very first line: `unit`. ModelMaker will emit this text exactly as is. The next token on that line is a macro, `<!UnitName!>`. I'll discuss macros in depth later; for now, understand that ModelMaker will expand the macro in-place. In this case, the macro represents the name of the unit, and that text will be emitted.

Finally, an example of a code-generation tag appears directly under the `type` keyword:

```
MMWIN:CLASSINTERFACE TDateForm; ID=37;
```

In this case, the tag tells ModelMaker to expand the class interface for `TDateForm` at this point in the unit code.

Thus, when editing code in the Unit Code Editor, you are looking at a hybrid of code managed by you and code managed by ModelMaker. Take care when editing this code not to disturb the ModelMaker-managed code unless you know what you're doing. It's analogous to editing code in a Delphi-managed DPR file—you can get in trouble fast if you're not careful. Nevertheless, this is where you would add a non-class-type declaration (an enumerated type, for instance). You would handle it just as you would in Delphi, adding the type declaration into the `type` section of the unit:

```
unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Dates, StdCtrls;

type
  TmyWeekday = (wdSunday, wdMonday, wdTuesday, wdWednesday,
      wdThursday, wdFriday, wdSaturday);

MMWIN:CLASSINTERFACE TDateForm; ID=37;
var
  DateForm: TDateForm;

implementation

{$R *.DFM}

MMWIN:CLASSIMPLEMENTATION TDateForm; ID=37;
end.
```

In this example, ModelMaker will emit the type declaration just as you've entered it—as plain text—and then begin expanding the `TDateForm` class declaration.

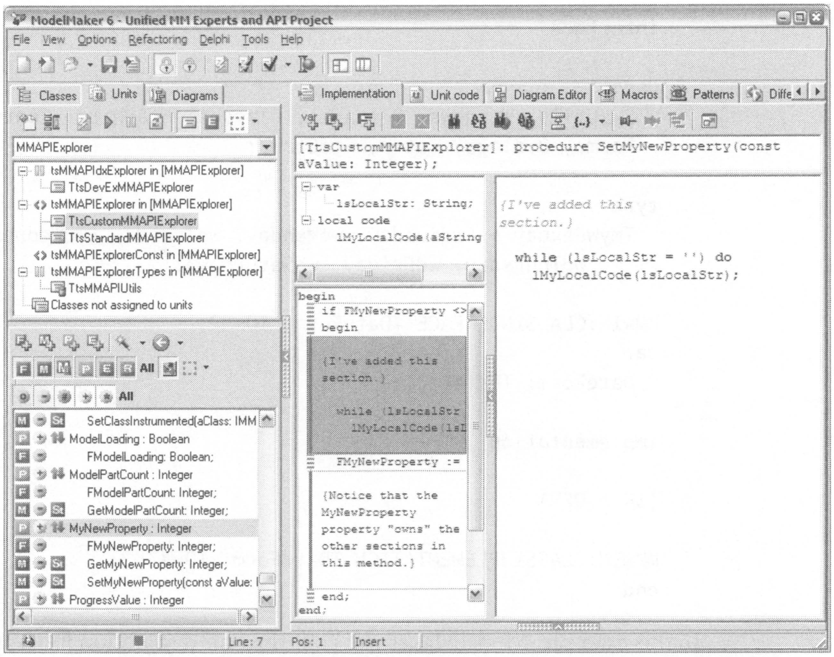
ModelMaker offers tools within the Unit Code Editor for managing unit-level methods, and they are a significant convenience when you have large routine library-type units. However, now that you're using ModelMaker, you can leverage its strong refactoring features to objectify some of those routines.

The Method Implementation Code Editor

ModelMaker's Method Implementation Code Editor (see Figure 11.7) is quite different from the Unit Code Editor. The editor takes up the right two thirds of the screen. In this example, I've added a fictional property named `MyNewProperty` and allowed ModelMaker to generate the state field and the read and write access methods. The write access method is active in the editor.

Next to the code editor on the right you can see two interesting windows. The tree view on top is the local code explorer: Here you can manage local variables and local procedures. Below that is the Section List; ModelMaker allows you to break up code within a method implementation into sections. In part, this is an organizational convenience; but more importantly, it allows ModelMaker to control specific sections of code. Just as ModelMaker can own parts of the model (like a property access method automatically generated for a property), it can also own sections of code within a method. Most often this occurs when you have chosen to have ModelMaker generate the read or write code within a property access method. Notice that in this example, the first, third, and fifth sections have a red and white dashed left margin, indicating that they are owned by ModelMaker. The sections with the green margin are user-owned. When ModelMaker generates this method, the code will be emitted in the order shown, one section after another.

FIGURE 11.7
ModelMaker with the Implementation tab active


ModelMaker 6 - Unified MM Experts and API Project
File View Options Refactoring Delphi Tools Help
Classes
Units
Diagrams
MMAPIExplorer
tsMMAPIdxExplorer in [MMAPIExplorer]
TtsDevExMMAPIExplorer
tsMMAPIExplorer in [MMAPIExplorer]
TtsCustomMMAPIExplorer
TtsStandardMMAPIExplorer
tsMMAPIExplorerConst in [MMAPIExplorer]
tsMMAPIExplorerTypes in [MMAPIExplorer]
TtsMMAPIUtils
Classes not assigned to units
All
All
SetClassInstrumented(aClass: IMM
ModelLoading : Boolean
FModelLoading: Boolean;
ModelPartCount : Integer
FModelPartCount: Integer;
GetModelPartCount: Integer;
MyNewProperty : Integer
FMyNewProperty: Integer;
GetMyNewProperty: Integer;
SetMyNewProperty(const aValue:
ProgressValue : Integer
Line: 7
Pos: 1
Insert
Implementation
Unit code
Diagram Editor
Macros
Patterns
Diffe

```
[TtsCustomMMAPIExplorer]: procedure SetMyNewProperty(const
aValue: Integer);
```

```
var
  lsLocalStr: String;
local code
  lMyLocalCode(aString
```

```
begin
  if FMyNewProperty <>
  begin

  {I've added this
  section.}

    while (lsLocalStr
      lMyLocalCode(lsL
    FMyNewProperty :=

  {Notice that the
  MyNewProperty
  property "owns" the
  other sections in
  this method.}

  end;
end;
```

```
{I've added this
section.}

  while (lsLocalStr = '') do
    lMyLocalCode(lsLocalStr);
```

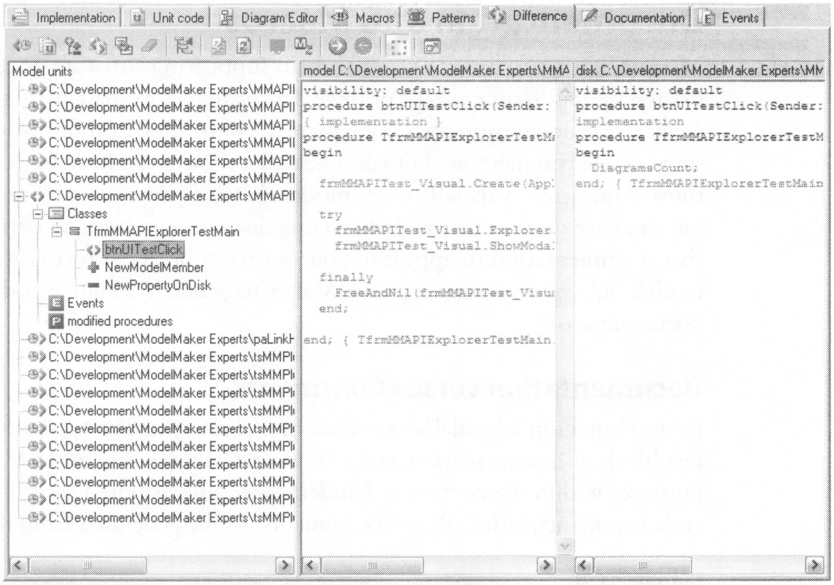
WARNING *A big drawback of writing code within ModelMaker Implementation windows is that is lacks any of the forms of Code Completion offered by the Delphi IDE.*

The Difference View

As I mentioned earlier, it's easy to get into a situation where your model is out of synch with your source files. If both your model and your source files have been edited, you can't simply regenerate the files from the model, lest you overwrite the source file changes. Likewise, you can't re-import the units, for fear of eliminating the changes to the model. Fortunately, ModelMaker offers one of the most robust differencing tools I've seen. When you find your model out of synch, it's time to visit the Difference tab (see Figure 11.8).

ModelMaker offers a variety of ways to view differences. You can view a standard text file comparison, time stamp differences, or even a comparison of two selected classes within the model. My favorite is on display in Figure 11.8—a structured difference. ModelMaker temporarily imports the source file on disk (thereby objectifying it just like the internal code model) and compares the imported file with the same unit and classes in the model on an object and attribute level rather than as text. The result is a much faster, much more accurate comparison.

FIGURE 11.8
ModelMaker's Difference view


Implementation
Unit code
Diagram Editor
Macros
Patterns
Difference
Documentation
Events
Model units
C:\Development\ModelMaker Experts\MMAPII
C:\Development\ModelMaker Experts\MMAPII
C:\Development\ModelMaker Experts\MMAPII
C:\Development\ModelMaker Experts\MMAPII
C:\Development\ModelMaker Experts\MMAPII
C:\Development\ModelMaker Experts\MMAPII
C:\Development\ModelMaker Experts\MMAPII
Classes
TfrmMMAPIExplorerTestMain
btnUITestClick
NewModelMember
NewPropertyOnDisk
Events
modified procedures
C:\Development\ModelMaker Experts\paLinkH
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI
C:\Development\ModelMaker Experts\tsMMPI

model C:\Development\ModelMaker Experts\MMA

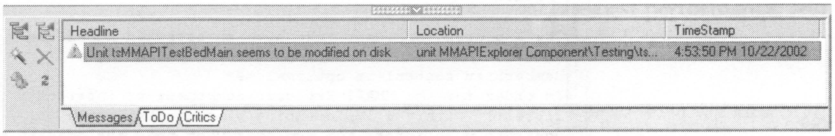
```
visibility: default
procedure btnUITestClick(Sender:
{ implementation }
procedure TfrmMMAPIExplorerTestMa
begin

  frmMMAPITest_Visual.Create(Appl

  try
    frmMMAPITest_Visual.Explorer.
    frmMMAPITest_Visual.ShowModal

  finally
    FreeAndNil(frmMMAPITest_Visua
  end;

end; { TfrmMMAPIExplorerTestMain.
```

disk C:\Development\ModelMaker Experts\MM

```
visibility: default
procedure btnUITestClick(Sender:
implementation
procedure TfrmMMAPIExplorerTestM
begin
  DiagramsCount;
end; { TfrmMMAPIExplorerTestMain
```

Notice in the tree view in Figure 11.8 the icons that denote differences. The red <> indicates that both the model and the source file contain the indicated class member (`btnUITestClick` in the example) but the two instances differ. The differing code is displayed in the memo controls to the right. The green + in the tree view indicates that the indicated class member exists only in the model, not on the disk. The blue - indicates that the class member exists only on disk, not in the model. With this information, you can choose how to proceed in re-synchronizing your model. One nifty feature is the ability to re-import a selected method (rather than the whole unit) from within the Difference view.

This approach implies that it is very important to know when your model is out of synch, so you don't override your changes on disk when regenerating the source files. Fortunately, ModelMaker offers several safeguards that can prevent this situation. One of these is a Design Critic (see the "Little-Known Tidbits" section later in the chapter). If your Design Critics are enabled, ModelMaker's Message View will warn you when a file on disk has changed:



Headline	Location	TimeStamp
Unit tsMMAPITestBedMain seems to be modified on disk	unit MMAPIExplorer Component\Testing\ts...	4:53:50 PM 10/22/2002

Messages
ToDo
Critics

The Event Types View

ModelMaker enables the management of event types on the Events tab; there you can edit event type declarations. But keep in mind that although a new event type may exist in ModelMaker's internal code model, it does not exist in a source file until you add the event type's declaration to a unit. The easiest way to manage that process is to drag the event type declaration from the Events view list to the Unit list and drop the item into a unit.

Documentation and Macros

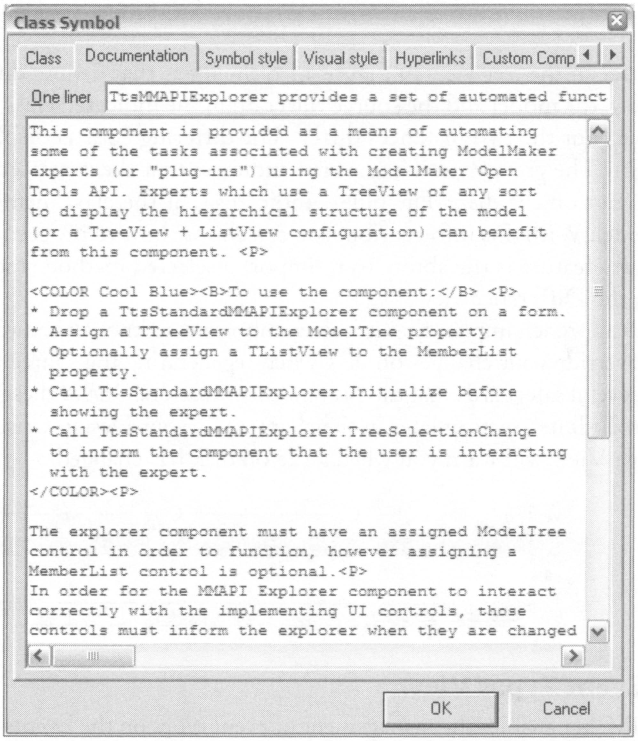
ModelMaker can come in very handy in supporting software documentation efforts. You need to master an important concept prior to proceeding (fortunately, it is not complex): Within ModelMaker, documentation does not equate to comments. Fear not; you can do complicated things with source code commenting, but you must take some steps to cause ModelMaker to emit (or import) those comments. Virtually every model element (classes, units, members, diagram symbols, and so on) can have documentation, but entering documentation for an element will not automatically cause that documentation to appear in your source code. That text is attached to the element within the model, but you must cause ModelMaker to generate a source code comment that contains your documentation.

Documentation versus Comments

Every element in a ModelMaker code model can own two types of documentation: a standard, large text block of documentation, and a short one liner (see Figure 11.9). These texts can serve multiple purposes within the context of ModelMaker and, as noted earlier, do not directly equate to source code comments, although such comments can be generated from them.

FIGURE 11.9

The Documentation tab of a Class Symbol



One liner: TtsMMAPIExplorer provides a set of automated funct

```
This component is provided as a means of automating
some of the tasks associated with creating ModelMaker
experts (or "plug-ins") using the ModelMaker Open
Tools API. Experts which use a TreeView of any sort
to display the hierarchical structure of the model
(or a TreeView + ListView configuration) can benefit
from this component. <P>

<COLOR Cool Blue><B>To use the component:</B> <P>
* Drop a TtsStandardMMAPIExplorer component on a form.
* Assign a TTreeView to the ModelTree property.
* Optionally assign a TListView to the MemberList
  property.
* Call TtsStandardMMAPIExplorer.Initialize before
  showing the expert.
* Call TtsStandardMMAPIExplorer.TreeSelectionChange
  to inform the component that the user is interacting
  with the expert.
</COLOR><P>

The explorer component must have an assigned ModelTree
control in order to function, however assigning a
MemberList control is optional.<P>
In order for the MMAPI Explorer component to interact
correctly with the implementing UI controls, those
controls must inform the explorer when they are changed
```

In this example, a class has both types of documentation. This documentation might appear in a diagram as an attached annotation (either the one liner or the standard documentation can be used), or you can specify that either or both be used as part of the source code file's comments. To do this, you'll need to use ModelMaker's powerful macros (discussed in the next section) and change some of your project options. For now, let's not worry about the macros (accepting the defaults) and look only at the project options.

Access the Project Options dialog by selecting Project Options from the Options menu and select the Source Doc Generation tab. Here you will find many options regarding the generation of source code comments from ModelMaker documentation. (For more information, see the online ModelMaker help.) To see source code commenting in action, select Before Declaration from the Method Implementation section of the In Source Documentation Generation group. Now any methods that contain documentation will use the default macro to generate source code comments.

ModelMaker can also import comments from a source unit and associate those comments with the appropriate code model elements. To do this, you must signify your comments with a Documentation Import Signature (see the Source Doc Import tab of the Project Options dialog) and tell ModelMaker what lines to import into the model. Thus if your method implementation has comments like the following, you can tell ModelMaker to ignore the first five lines and import only the actual comment text:

```
{*****************************************************
TmyClass.DoSomething
Returns:   String
Visibility:  Public
Comment:
    This is the actual comment that we want ModelMaker to import. The
    first 5 lines of this comment block should not be imported into
    the model.}
```

When you're configuring ModelMaker for source commenting, it's important to watch out for *comment creep*. This can occur when your comment import and export settings do not quite match. For instance, if the macro controlling your source comment output adds six lines to the comment before adding the documentation text, but your import settings eliminate only five lines, then each import/generation cycle will add a redundant line of text to your comment.

Working with Macros

Macros represent one of the key features of ModelMaker: They are easy to learn, but difficult to master. A macro is an identifier that represents a block of text. When ModelMaker encounters a macro, it attempts to replace the macro name with the text the macro represents.

You've seen this process in action in the Unit Code Editor: `<!UnitName!>` is replaced at code generation time with the name of the unit being generated. This is an example of an entity-specific macro that is always different depending on what unit is being generated. The macro, `UnitName`, is predefined, but the result will differ by context.

ModelMaker includes many predefined macros (the complete list is on page 75 of the User Manual, the usermanual620.pdf file you can find on Delphi's Companion CD). You can create your own macros of varying complexity (even nested macros) in the Macros tab. You can also *override* certain predefined documentation expander macros. For instance, if you enable method implementation documentation but supply no macro, ModelMaker will use its built-in macro to generate the comments. However, if you declare and define a macro named `MemberImpDoc`, ModelMaker will use this macro when generating method comments. (See ModelMaker's online help for a list of override-able macros used for source comment generation, looking for the topic "Documentation Macros.")

Macros are not used only at code-generation time. You can also expand macros while typing within a code editor. In this case, you can parameterize a macro so that when ModelMaker attempts to expand it, you will be prompted for values. These values can be inserted into the text being expanded.

Refactoring Your Code

Refactoring is one of those trendy terms in computer programming that is constantly bandied about, but that means different things to different people. Refactoring is basically the process of improving your existing code in place without altering its external behavior. There is no single refactoring process to which you must adhere—it's simply the task of trying to improve your code in place without breaking too much around it.

Many texts are dedicated to the concept of refactoring, so I will simply look at some of the specific ways ModelMaker can assist you in refactoring your code. Again, ModelMaker's internal code model plays a big role—remember that developing in ModelMaker is not just development of object-oriented code; it's also a development process that is assisted by object orientation. Because all these code model elements are stored internally as objects—objects that have references to each other—and because source code units are completely regenerated from this model every time you choose to generate your code, any changes to the code elements are propagated throughout your classes instantly.

The perfect example is again a class property. If you have a property named `MyNewProperty` with attending read/write methods (maintained by ModelMaker and named `GetMyNewProperty` and `SetMyNewProperty`), and you would like to rename the property `MyProperty`, doing so requires only one step: rename the property. ModelMaker takes care of the rest—the access methods are automatically renamed `GetMyProperty` and `SetMyProperty`. If the property appears in a diagram, the diagram is automatically updated to represent the change. (One caveat: ModelMaker will not automatically search your code for instances of `MyNewProperty`—you'll have to do this with a global search and replace within ModelMaker.) This is a simple example, but it illustrates how ModelMaker simplifies the task of refactoring; as you move and rename code elements, ModelMaker will handle the majority of the details for you. Now let's look at some specifics:

Simple Renaming This task is quite simple and we've already touched on it, but its usefulness cannot be overstressed. Code model element name changes are propagated by ModelMaker through the code model to all instances of that element of which it is aware.

Reparenting Classes This absurdly simple process can be done a number of different ways. Most commonly, you can simply drag a class in the Classes view from one parent node to another (you can also do this in a class diagram by dragging the generalization arrow from the old parent to the new parent)—now your class has a new parent. If inheritance is restricted, ModelMaker will automatically update the child class's inherited methods to match the new parent's declarations. The next time you generate your code, these changes will automatically appear.

Moving Classes between Units It is also simple to move a class to a new unit. In the Units view, drag the class from its current place to the new unit. All relevant code (declarations, implementations, and comments) will be regenerated in the new unit.

Moving Members between Classes In refactoring, this process is known as "moving features (or responsibilities) between objects." The idea is simple: as development progresses, you may discover that certain responsibilities (implemented as class members) are more appropriately moved to another class. You can do sousing drag and drop. Select the desired class members from the member list and drag them to the new class in the Classes (hold down the Shift key to move rather than copy).

Converting Members This is one of ModelMaker's niftier refactoring features. Right-clicking on a member in the Member List will display the context menu that contains the Convert To menu item and subitems. Selecting one of these subitems lets you convert an existing class member from one member type to another. For instance, if you have a private field named `FMyInteger`, and you opt to convert it to a property, ModelMaker automatically creates a public property named `MyInteger`, which reads from and writes to `FMyInteger`. Likewise, you can convert this field into a method—it will be a private function named `MyInteger` that returns an integer.

Restricted Inheritance In the method editor dialog is an Inheritance Restricted check box. When this box is checked, ModelMaker does not allow you to change most of the method's attributes, because those attributes are being set based on the ancestor class's implementation of the overridden method. If you change the declaration of a method in an ancestor class, those changes will automatically be applied in any descendant classes where the overriding method is set to restricted inheritance.

If you have experience in refactoring (or you are familiar with the latest versions of JBuilder), this may not seem like a particularly impressive set of refactoring tools. However, when compared to what is possible in Delphi alone, this is a wonderful collection of possibilities. In addition, ModelMaker's OpenTools API gives you access to most of the code model. If you're discontented with what ModelMaker offers out of the box, you can extend its refactoring capabilities on your own.

NOTE *Additionally, I can clandestinely say that I've seen beta versions (at the time of this writing) of a future release of ModelMaker that contain sets of new refactoring tools. Most of them are pulled straight from Martin Fowler's book on refactoring, and they are impressive.*

Applying Design Patterns

ModelMaker puts it all together impressively in its support of design patterns. (A full discussion of design patterns is beyond the scope of this chapter; however, if you're unfamiliar with patterns, see the sidebar "Design Patterns 101" for a short introduction.) ModelMaker provides the convenience of applying a pattern implementation with a single mouse click. Depending on the pattern you select, a variety of actions take place—some patterns display a wizard before adding code, and others simply add their members to the selected class. As I've discussed earlier, these new members are owned by ModelMaker and are easily updated as a result. In addition, should you choose to un-apply the pattern, ModelMaker removes any class members it added for that pattern.

As an example, let's look at the Singleton pattern. Suppose you have a class and you want only one instance of it at most. Here is the sample class:

```
type
  TOneTimeData = class (TObject)
  private
    FGlobalCount: Integer;
    procedure SetGlobalCount(const Value: Integer);
  public
    property GlobalCount: Integer read FGlobalCount write SetGlobalCount;
  end;
```

The Singleton pattern mandates the use of a single entry point (the class function `Instance` in the ModelMaker implementation of this pattern, as seen next) to gain access to your single instance of the class. If the instance does not yet exist, it will be created and returned; otherwise the existing instance will be returned. Because `Instance` is the entry point, you'll disallow the use of `Create` for this class. Once you apply the Singleton pattern to your class in ModelMaker, it appears thus:

```
type
  TOneTimeData = class (TObject)
  private
    FGlobalCount: Integer;
    procedure SetGlobalCount(const Value: Integer);
  protected
    constructor CreateInstance;
    class function AccessInstance(Request: Integer): TOneTimeData;
  public
    constructor Create;
    destructor Destroy; override;
    class function Instance: TOneTimeData;
    class procedure ReleaseInstance;
    property GlobalCount: Integer read FGlobalCount write SetGlobalCount;
  end;
```

I won't list the method implementations here; you can look them up either by applying the pattern yourself, or by looking up the source code of the PatternDemo example.

WARNING *The code used by ModelMaker to implement the Singleton patterns is based on the interesting use of a constant within a method to mimic per-class data. However, this code fails to compile unless you enable the Assignable Typed Constants Delphi compiler option, which is disabled by default.*

DESIGN PATTERNS 101

Whereas programmers concentrate on implementing specific classes, designers are more focused on making different classes/objects work together. Although it is difficult to give a precise definition of software design, essentially it is the organization of the overall structure of a program.

Looking at different people's design solutions to different problems, you can notice similarities and common elements. A *pattern* is the acknowledgement of such a common design, expressed in a standard way, and abstracted enough to be applicable in a number of different situations. Design patterns relate to design reuse more than code reuse. Although the coded solution of a pattern can be an inspiration to the programmer,

Continued on next page

DESIGN PATTERNS 101 *(continued)*

the actual focus is in the design: Even if you might have to rewrite the code, starting with a clear and proven design will save you considerable time. Design patterns do not cover primitive building blocks (like a hash table or a linked list) or domain-specific problems (like analysis patterns do).

The recognized originator of the pattern movement was not a software designer but an architect, who noticed the use of patterns in buildings. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice." Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides wrote *the* book that started the pattern movement in the software world: *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995). The authors are often indicated as "Gamma et al.," but are more frequently called the Gang of Four or simply GoF. The book is often colloquially referenced as the "GoF book."

The book describes the notion of software patterns, indicates a precise way of describing them, and provides a catalog of 23 patterns divided into three groups: creational, structural, and behavioral. Most of the GoF patterns are implemented in C++, and some in Smalltalk, although they generally abstract the language and are equally applicable in Java or Delphi.

The core structure of a pattern is the following:

- The *pattern name* is important, so you can refer to the pattern when talking to other programmers and designers.
- The *problem* describes when to apply a pattern, eventually including context and conditions.
- The *solution* describes the elements part of the design and their relationships. It isn't an implementation, only an abstract description of class responsibilities and collaborations.
- The *consequences* are the results and trade-off of applying a pattern, including space and time constraints.

Currently, no book is devoted to patterns from the Delphi perspective. However, many articles have appeared in the Delphi magazines (including *Delphi Informant* and *The Delphi Magazine*). The classic GoF patterns have been a source of inspiration for many articles, along with the detailed discussion of patterns available with the ModelMaker documentation (see the website to download it).

I don't always agree with the Delphi implementation of some standard patterns. In fact, I tend to focus on the underlying design and how it can be preserved while moving the GoF implementation (often in C++ or Java) to Delphi and leveraging its specific language features. Other authors tend to port the code, which is *only* a way to implement the design. Learning about patterns is important, because you'll develop a common jargon with other programmers and learn better ways to apply OOP techniques (particularly encapsulation and low coupling).

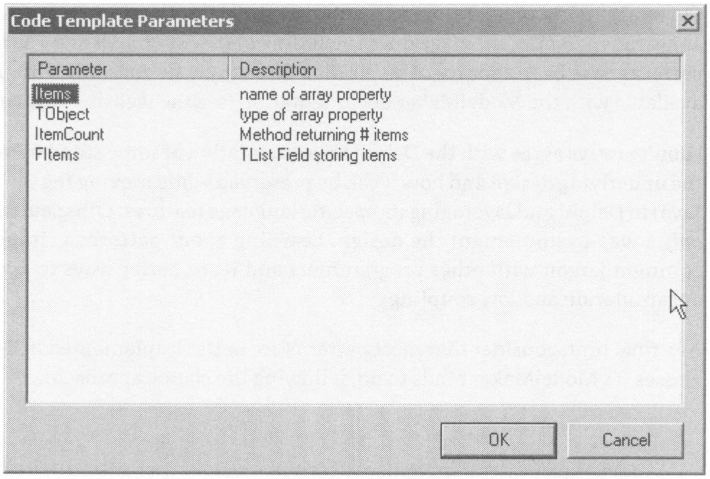
As a final hint, consider that most patterns are better implemented in Delphi using interfaces rather than classes (as ModelMaker tends to do, following the classic approach).

ModelMaker offers implementations of several other patterns, including Visitor, Observer, Wrapper, Mediator, and Decorator. They are hard-coded within ModelMaker to be applied a specific way, and some of the implementations are better than others. This has been a point of contention among some developers, and for that reason (among others) ModelMaker supports another means of applying patterns: code templates (discussed in the next section). This approach allows creation and customization on the part of the developer. However, don't overlook ModelMaker's extant support for patterns; they're quite good and offer a fixed, solid, working, Delphi-specific implementation of these common problems.

Code Templates

Yet another powerful feature in ModelMaker (which is seemingly lost, tucked in among the myriad other conveniences) is code templates, a technique you can use to create your own implementations of design patterns. Code templates are like a snapshot of part of a class that can be applied to another class. In other words, it's a collection of class members, saved in a template that can be added to another class, en masse. Better still, these templates can be parameterized (much like macros) so that when you apply one to a class, a dialog will pop up to ask you to fill in certain values, which are then applied as part of the template.

One example is an array property. Declaring an array property is simple in ModelMaker, but completely implementing one requires several steps: You must have not only the array property itself, but also a `TList` or descendant to contain the array elements, and a means of supplying a count of the stored elements. Even for this simple example, it takes a bit of work to get your array property up and running. Enter the array property template. Open a model in ModelMaker (or create a new model and add a `TObject` descendant) and select a class to which you'd like to add your new array property. Right-click in the Member List and select Code Templates. There should now be a floating toolbar named Code Templates (note that this same toolbar is available in the Patterns tab). Click the Apply Array Property Template button to open the Code Template Parameters dialog. It contains a list of items you can specify for the template that is about to be applied, as you can see in Figure 11.10. You can highlight any item in the left column and press F2 to edit the value for that parameter. Accept the defaults and click OK.



Parameter	Description
Items	name of array property
TObject	type of array property
ItemCount	Method returning # items
FItems	TList Field storing items

FIGURE 11.10 ModelMaker's Code Template Parameters dialog box

Your class should now contain the following members:

```
private
  FItems: TList;
protected
  function GetItemCount: Integer;
  function GetItems(Index: Integer): TObject;
public
  property ItemCount: Integer read GetItemCount;
  property Items[Index: Integer]: TObject read GetItems;
```

You can see how flexible this technique can be. Other common tasks (like strongly typed lists) and your own implementations of design patterns are easily implemented; let's see how.

To create your own code template, begin with an existing class that already has the members you wish to turn into a template. Select that class, and then, in the Member List, select the members you wish to use (these can be any type of member). Right-click in the Member List and select Create Code Template; the Save Code Template dialog will appear. It is much like a standard Save As dialog (and you *do* specify where to save the template), but you can also detail how you'd like the template to appear. Specify a name for the template and on which page of the template palette you wish it to appear. Take note of the resulting confirmation message; you can change the palette bitmap if you wish.

Your new template is now available in the template palette; you can add this template to any class. To parameterize your template, you must alter the PAS file that was created when you saved the template. For example, here is part of the `ArrayProp_List.pas` file used for the Array Property template:

```
unit ArrayProp_List;

//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

  TCodeTemplate = class (TObject)
  private
    <!FItems!>: TList;
  protected
    function Get<!ItemCount!>: Integer;
    function Get<!Items!>(Index: Integer): <!TObject!>;
  public
    property <!ItemCount!>: Integer read Get<!ItemCount!>;
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
  end;
```

Notice the lines that begin with `//DEFINEMACRO`. This is where you declare your parameters; they will appear in the Code Template Parameters dialog you saw earlier. Each line is a Name/Value pair: the element on the left of the = is the editable value, and the element on the right is a description you can provide to explain the parameter.

After you supply a list of parameters, they can be used as macros in your template code. Note in the example lines like this:

```
property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
```

When this property is added to a class as part of the template, the macros (things like `<!Items!>`) will be replaced with the value of the appropriate parameter. In this way, you can use parameters to deeply customize your code templates.

Little-Known Tidbits

As a parting thought, here's a list of interesting features you might want to examine more closely:

Rethink Orthogonal You can change the default straight, diagonal lines in the Diagram Editor to orthogonal lines by pressing Ctrl+O. You can also force ModelMaker to try to find the shortest possible orthogonal path for a line by pressing Shift+Ctrl+O.

Visual Styles Manager This manager (available in the shortcut menu of the diagram view, under Visual style ➢ Style manager) deserves an entire section. Take some time to check it out. You can define a wide variety of hierarchically related visual styles for your diagram symbols and apply them on the fly. Also, don't forget to click the Use Printing Style button in the Diagram Editor to strip out the non-printing elements and see what the diagram will look like on paper.

Design Critics Design critics are an impressive QA feature in ModelMaker. They are little proofreaders running the background, checking out your code. To get at them, make sure Show Messages is enabled (Shift+Ctrl+M), right-click the Message view, and select Show Critics Manager. I advise against turning off the time-stamp-checking design critic, because it will warn you if the source file on the disk has changed outside of ModelMaker. You can also create your own design critics via the ModelMaker OpenTools API.

Creational Wizard This is yet another nifty bit of automation for the busy Delphi programmer. The Creational Wizard (available from the Wizards button in the Member List) checks the model for class members that need to be instantiated or freed and adds them to the appropriate constructor or destructor. It will do a few other things too, and there are some caveats; press F1 while you're in the wizard to access the online help.

Open Tools API Much like in Delphi's Tools API, this ModelMaker feature allows the creation of plug-in experts for ModelMaker. The API is robust and includes access to diagrams as well as the entire code model. The possibilities for extending ModelMaker in this way are extreme.

What's Next?

This chapter has provided a limited overview of the capabilities of ModelMaker, covering apparently unrelated topics like UML diagrams, patterns, refactoring, and developer's documentation. There was a reason to touch on all these aspects, of course: ModelMaker can help you with all of them, whereas the Delphi IDE by itself provides little or no support for these features.

Refer to the websites and other documentation mentioned to gain a better understanding of the various techniques. As mentioned, ModelMaker provides a lot of documentation, including material about diagrams and patterns. Visit the product website `www.modelmakertools.com` to download more than you'll find in the default installation.

If you want more information about the techniques I've discussed, or just how to get started, refer to the ModelMaker online help and the User's Manual (available on Delphi 7's Companion CD). I also recommend the Thoughtsmithy website, where Robert Leahey has authored a set of "Getting Started with ModelMaker" tutorials. You can find them at:

`www.thoughtsmithy.com/mmjump/MMGettingStarted_Intro.html`

Next up is another chapter devoted to Delphi applications architecture. Chapter 12 features an in-depth analysis of COM-related technologies available in the Windows operating system and fully supported by Delphi. After that, we'll delve into the database-programming portion of the book.

Chapter 12

From COM to COM+

FOR ABOUT 10 YEARS, starting soon after the release of Windows 3.0, Microsoft has been promising that its operating system and its API would be based on a real object model instead of functions. According to the speculations, Windows 95 (and later Windows 2000) should have been based on this revolutionary approach. Nothing like this happened, but Microsoft kept pushing COM (Component Object Model), built the Windows 95 shell on top of it, pushed applications integration with COM and derivative technologies (such as Automation), and reached the peak by introducing COM+ with Windows 2000.

Now, soon after the release of the complete foundation required for high-level COM programming, Microsoft has decided to switch to a new core technology, part of the .NET initiative. My impression is that COM wasn't really suited for the integration of fine-grained objects, although it succeeded in providing an architecture for integrating applications or large objects.

In this chapter, you'll build your first COM object; I'll stick to the basic elements to let you understand the role of this technology without delving heavily into the details. We'll continue by discussing Automation and the role of type libraries, and you'll see how to work with Delphi data types in Automation servers and clients.

In the final part of the chapter, we'll explore the use of embedded objects, with the OleContainer component, and the development of ActiveX controls. I'll also introduce stateless COM (MTS and COM+) technologies and a few other advanced ideas including the .NET integration support offered by Delphi 7.

This chapter covers the following:

- What is COM?
- COM, GUIDs, and class factories
- Delphi interfaces and COM
- The VCL COM-support classes
- Creating and using Automation servers
- Using type libraries

- The Container component
- Building an ActiveX and an ActiveForm
- Introducing COM+
- COM and .NET in Delphi 7

A Short History of OLE and COM

Part of the confusion related to COM technology comes from the fact that Microsoft used different names for it in the early years for marketing reasons. Everything began with Object Linking and Embedding (OLE, for short), which was an extension of the DDE (Dynamic Data Exchange) model. Using the Clipboard allows you to copy raw data, and using DDE allows you to connect parts of two documents. OLE lets you copy data from a server application to a client application, along with information regarding the server or a reference to information stored in the Windows Registry. The raw data might be copied along with the link (object embedding) or kept in the original file (object linking). OLE documents are now called *active documents*.

Microsoft updated OLE to OLE 2 re-implementing it not to be just an extension to DDE and began adding new features, such as OLE Automation and OLE Controls. The next step was to build the Windows 95 shell using OLE technology and interfaces and then to rename the OLE Controls (previously known also as OCX) as ActiveX controls, changing the specification to allow for lightweight controls suitable for distribution over the Internet. For a while, Microsoft promoted ActiveX controls as suitable for the Internet, but the idea was never fully accepted by the development community—certainly not as "suitable" for Internet development.

As this technology was extended and became increasingly important to the Windows platform, Microsoft changed the name back to OLE, and then to COM, and finally to COM+ for Windows 2000. These changes in naming were only partially related to technological changes and were driven to a large extent by marketing purposes.

What, then, is COM? Basically, the Component Object Model is a technology that defines a standard way for a client module and a server module to communicate through a specific interface. Here, *module* indicates an application or a library (a DLL); the two modules may execute on the same computer or on different machines connected via a network. Many interfaces are possible, depending on the role of the client and server, and you can add new interfaces for specific purposes. These interfaces are implemented by server objects. A server object usually implements more than one interface, and all the server objects have a few common capabilities, because they must all implement the `IUnknown` interface (which corresponds to the Delphi-specific `IInterface` I introduced in Chapter 2, "The Delphi Programming Language").

The good news is that Delphi is fully compliant with COM. When Delphi 3 came out its COM implementation was much easier and integrated in the language than than C++ or other languages were at that time, up to the point that even programmers on the Windows R&D team commented "we should have done COM the way Delphi does COM." This simplicity mainly derives from the incorporation of interface types into the Delphi language. (Interfaces are also used in a similar way to integrate Java with COM on the Windows platform.)

As I've mentioned, the purpose of COM interfaces is to communicate between software modules, which can be executable files or DLLs. Implementing COM objects in DLLs is generally simpler, because in Win32, a program and the DLL it uses reside in the same memory address space. This means that if the program passes a memory address to the DLL, the address remains valid. When you use two executable files, COM has a lot of work to do behind the scenes to let the two applications communicate. This mechanism is called *marshaling* (which, to be accurate, is required also by DLLs if the client is multi-threaded). Note that a DLL implementing COM objects is described as an *in-process* server, whereas when the server is a separate executable, it is called an *out-of-process* server. However, when DLLs are executing on another machine (DCOM) or inside a host environment (MTS), they are also out-of-process.

Implementing *IUnknown*

Before we begin looking at an example of COM development, I'll introduce a few COM basics. Every COM object must implement the `IUnknown` interface, also dubbed `IInterface` in Delphi for non-COM usage of interfaces (as you saw in Chapter 2). This is the base interface from which every Delphi interface inherits, and Delphi provides a couple of different classes with ready-to-use implementations of `IUnknown`/`IInterface`, including `TInterfacedObject` and `TComObject`. The first can be used to create an internal object unrelated to COM, and the second is used to create objects that can be exported by servers. As you'll see later in this chapter, several other classes inherit from `TComObject` and provide support for more interfaces, which are required by Automation servers or ActiveX controls.

As mentioned in Chapter 2, the `IUnknown` interface has three methods: `_AddRef`, `_Release`, and `QueryInterface`. Here is the definition of the `IUnknown` interface (extracted from the System unit):

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

The `_AddRef` and `_Release` methods are used to implement reference counting. The `QueryInterface` method handles the type information and type compatibility of the objects.

NOTE *In the previous code, you can see an example of an* `out` *parameter, a parameter passed back from the method to the calling program but without an initial value passed by the calling program to the method. The* `out` *parameters have been added to the Delphi language to support COM, but they can as well be used in a normal application, as in certain circumstances this makes parameters passing more efficient (significant cases are those of interfaces, strings, and dynamic arrays). It's also important to note that although Delphi's language definition for the interface type is designed for compatibility with COM, Delphi interfaces do not require COM. This was already highlighted in Chapter 2, where I built an interface-based example with no COM support.*

You don't usually need to implement these methods, because you can inherit from one of the Delphi classes already supporting them. The most important class is `TComObject`, defined in the ComObj unit. When you build a COM server, you'll generally inherit from this class.

`TComObject` implements the `IUnknown` interface (mapping its methods to `ObjAddRef`, `ObjQueryInterface`, and `ObjRelease`) and the `ISupportErrorInfo` interface (through the `InterfaceSupportsErrorInfo` method). Notice that the implementation of reference counting for the `TComObject` class is thread-safe, because it uses the `InterlockedIncrement` and `InterlockedDecrement` API functions instead of the plain `Inc` and `Dec` procedures.

As you would expect if you remember the discussion of reference counting in Chapter 2, the `_Release` method of `TInterfacedObject` destroys the object when there are no more references to it. The `TComObject` class does the same. Also keep also in mind that Delphi automatically adds the reference-counting calls to the compiled code when you use interface-based variables, including COM variables.

Finally, notice that the role of the `QueryInterface` method is twofold:

- `QueryInterface` is used for type checking. The program can ask an object the following questions: Are you of the type I'm interested in? Do you implement the interface and the specific methods I want to call? If the answers are no, the program can look for another object, maybe asking another server.
- If the answers are yes, `QueryInterface` usually returns a pointer to the object, using its reference output parameter (`Obj`).

To understand the role of the `QueryInterface` method, it is important to keep in mind that a COM object can implement multiple interfaces, as the `TComObject` class does. When you call `QueryInterface`, you ask for one of the possible interfaces of the object, using the `TGUID` parameter.

In addition to the `TComObject` class, Delphi includes several other predefined COM classes. Here is a list of the most important COM classes of the Delphi VCL, which you'll use in the following sections:

- `TTypedComObject`, defined in the ComObj unit, inherits from `TComObject` and implements the `IProvideClassInfo` interface (in addition to the `IUnknown` and `ISupportErrorInfo` interfaces already implemented by the base class, `TComObject`).
- `TAutoObject`, defined in the ComObj unit, inherits from `TTypedComObject` and implements also the `IDispatch` interface.
- `TActiveXControl`, defined in the AxCtrls unit, inherits from `TAutoObject` and implements several interfaces (`IPersistStreamInit`, `IPersistStorage`, `IOleObject`, and `IOleControl`, to name just a few).

Globally Unique Identifiers

The `QueryInterface` method has a parameter of the `TGUID` type. This type represents a unique ID used to identify COM object classes (in which case the GUID is called CLSID), interfaces (in which case you'll see the term IID), and other COM and system entities. When you want to know whether an object supports a specific interface, you ask the object whether it implements the interface that has a given IID (which for the default COM interfaces is determined by Microsoft). Another ID is used to indicate a specific class or CLSID. The Windows Registry stores this CLSID with indications of the related DLL or executable file. The developers of a COM server define the class identifier.

Both of these IDs are known as GUIDs, or *globally unique identifiers.* If each developer uses a number to indicate its COM server, how can we be sure these values are not duplicated? The short answer is that we cannot. The real answer is that a GUID is such a long number (16 bytes, or 128 bits—a number with 38 digits!) that it is almost impossible to come up with two random numbers having the same value. Moreover, programmers should use the specific API call `CoCreateGuid` (directly or through their development environment) to come up with a valid GUID that reflects some system information.

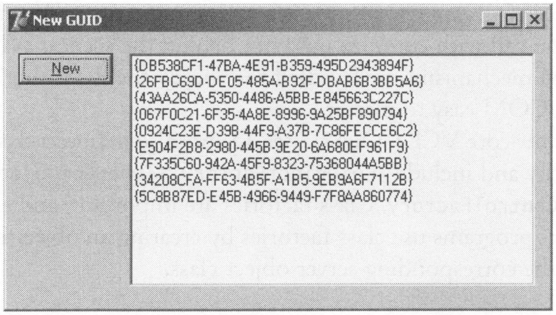
GUIDs created on machines with network cards are guaranteed to be unique, because network cards contain unique serial numbers that form a base for the GUID creation. GUIDs created on machines with CPU IDs (such as the Pentium III) should also be guaranteed unique, even without a network card. With no unique hardware identifier, GUIDs are unlikely to ever be duplicated.

WARNING *Be careful not to copy the GUID from someone else's program (which can result in two different COM objects using the same GUID). You should also not make up your own ID by entering a casual sequence of numbers. To avoid any problem, press Ctrl+Shift+G in the Delphi editor, and you will get a new, properly defined, truly unique GUID.*

In Delphi, the `TGUID` type (defined in the System unit) is a record structure, which is quite odd but required by Windows. Thanks to some Delphi compiler magic, typically set up to help make more straightforward some tedious or time consuming task, you can assign a value to a GUID using the standard hexadecimal notation stored in a string, as in this code fragment:

```
const
  Class_ActiveForm1: TGUID = '{1AFA6D61-7B89-11D0-98D0-444553540000}';
```

You can also pass an interface identified by an IID where a GUID is required, and again Delphi will magically extract the referenced IID. If you need to generate a GUID manually and not in the Delphi environment, you can call the `CoCreateGuid` Windows API function, as demonstrated by the NewGuid example (see Figure 12.1). This example is so simple that I've decided not to list its code.



```
{DB538CF1-47BA-4E91-B359-495D2943894F}
{26FBC69D-DE05-485A-B92F-DBAB6B3BB5A6}
{43AA26CA-5350-4486-A5BB-E845663C227C}
{067F0C21-6F35-4A8E-8996-9A25BF890794}
{0924C23E-D39B-44F9-A37B-7C86FECCE6C2}
{E504F2B8-2980-445B-9E20-6A680EF961F9}
{7F335C60-942A-45F9-8323-75368044A5BB}
{34208CFA-FF63-4B5F-A1B5-9EB9A6F7112B}
{5C8B87ED-E45B-4966-9449-F7F8AA860774}
```

FIGURE 12.1
An example of the GUIDs generated by the NewGuid example. Values depend on my computer and the time I run this program.

To handle GUIDs, Delphi provides the `GUIDToString` function and the opposite `StringToGUID` function. You can also use the corresponding Windows API functions, such as `StringFromGuid2`, but in this case, you must use the WideString type instead of the string type. Any time COM is involved, you have to use the WideString type, unless you use Delphi functions that automatically do the required conversion for you. When you need to bypass Delphi functions that can call COM API functions directly, you can use the `PWideChar` type (pointer to null-terminated arrays of wide characters) or casting a WideString to `PWideChar` (exactly as you cast a string to the `PChar` type when calling a low-level Windows API.) does the trick.

The Role of Class Factories

When have registered the GUID of a COM object in the Registry, you can use a specific API function to create the object, such as the `CreateComObject` API:

```
function CreateComObject (const ClassID: TGUID): IUnknown;
```

This API function will look into the Registry, find the server registering the object with the given GUID, load it, and, if the server is a DLL, call the `DLLGetClassObject` method of the DLL. This is a function every in-process server must provide and export:

```
function DllGetClassObject (const CLSID, IID: TGUID;
  var Obj): HResult; stdcall;
```

This API function receives as parameters the requested class and interface, and it returns an object in its reference parameter. The object returned by this function is a *class factory*.

As the name suggests, a class factory is an object capable of creating other objects. Each server can have multiple objects. The server exposes a class factory for each of the COM objects it can create. One of the many advantages of the Delphi simplified approach to COM development is that the system can provide a class factory for you. For this reason, I didn't add a custom class factory to my example.

The call to the `CreateComObject` API doesn't stop at the creation of the class factory, however. After retrieving the class factory, `CreateComObject` calls the `CreateInstance` method of the `IClassFactory` interface. This method creates the requested object and returns it. If no error occurs, this object becomes the return value of the `CreateComObject` API.

By setting up this mechanism (including the class factory and the `DLLGetClassObject` call), Delphi makes it simple to create COM objects. At the same time, Window's `CreateComObject` is just a simple function call with complex behavior behind the scenes. What's great in Delphi is that many complex COM mechanisms are handled for you by the RTL. Let's begin looking in detail at how Delphi makes COM easy to master.

For each of the core VCL COM classes, Delphi also defines a class factory. The class factory classes form a hierarchy and include `TComObjectFactory`, `TTypedComObjectFactory`, `TAutoObjectFactory`, and `TActiveXControlFactory`. Class factories are important, and every COM server requires them. Usually Delphi programs use class factories by creating an object in the initialization section of the unit defining the corresponding server object class.

A First COM Server

There is no better way to understand COM than to build a simple COM server hosted by a COM server DLL. A library hosting a COM object is indicated in Delphi as an ActiveX library. For this reason, you can begin the development of this project by selecting File ➢ New ➢ Other, moving to the ActiveX page, and selecting the ActiveX Library option. Doing so generates a project file I saved as FirstCom among the book demos. Here is its complete source code:

```
library FirstCom;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

The four functions exported by the DLL are required for COM compliance and are used by the system as follows:

- To access the class factory (`DllGetClassObject`)
- To check whether the server has destroyed all its objects and can be unloaded from memory (`DllCanUnloadNow`)
- To add or remove information about the server in the Windows Registry (`DllRegisterServer` and `DllUnregisterServer`)

You generally don't have to implement these functions, because Delphi provides a default implementation in the ComServ unit. For this reason, in the server code, you only need to export them.

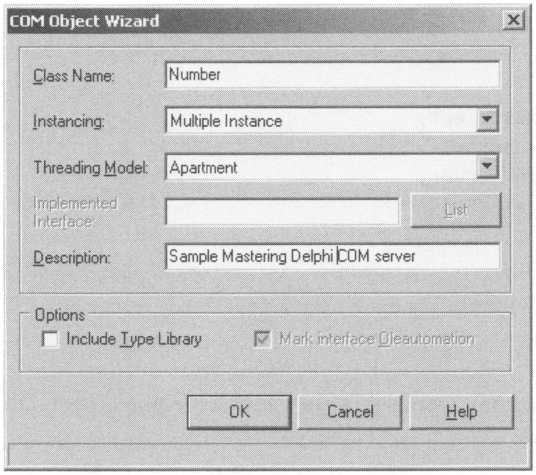
COM Interfaces and Objects

Now that the structure of your COM server is in place, you can begin developing it. The first step is to write the code of the interface you want to implement in the server. Here is the code of a simple interface, which you should add to a separate unit (called NumIntf in the example):

```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

After declaring the custom interface, you can add the object to the server. To accomplish this, you can use the COM Object Wizard (available in the ActiveX page of the File ➢ New ➢ Other dialog box). You can see this wizard's dialog box in Figure 12.2. Enter the name of the server's class and a description. I've disabled the generation of the type library (in which case the wizard disables the interface field in Delphi 7, different from what happened in Delphi 6) to avoid introducing too many topics at once. You should also choose an instancing and a threading model, as described in the related sidebar.

FIGURE 12.2
The COM Object Wizard



Class Name: Number
Instancing: Multiple Instance
Threading Model: Apartment
Implemented Interface:
Description: Sample Mastering Delphi COM server
☐ Include Type Library
☑ Mark interface Oleautomation

The code generated by the COM Object Wizard is quite simple. The interface contains the definition of the class to fill with methods and data:

```
type
  TNumber = class(TComObject, INumber)
    protected
      {Declare INumber methods here}
    end;
```

Beside the GUID for the server (saved in the `Class_Number` constant), there is also code in the `initialization` section of the unit, which uses most of the options you've set up in the wizard's dialog box:

```
initialization
  TComObjectFactory.Create(ComServer, TNumber, Class_Number, 'Number',
    'Number Server', ciMultiInstance, tmApartment);
```

This code creates an object of the `TComObjectFactory` class, passing as parameters the global `ComServer` object, a class reference to the class you've just defined, the GUID for the class, the server name, the server description, and the instancing and threading models you want to use.

The global `ComServer` object, defined in the ComServ unit, is a manager of the class factories available in the server library. It uses its own `ForEachFactory` method to look for the class supporting a given COM object request, and it keeps track of the number of allocated objects. As you've already seen, the ComServ unit implements the functions required by the DLL to be a COM library.

Having examined the source code generated by the wizard, you can now complete it by adding to the `TNumber` class the methods required for implementing the `INumber` interface and write their code, and you'll have a working COM object in your server.

COM INSTANCING AND THREADING MODELS

When you create a COM server, you should choose a proper instancing and threading model, which can significantly affect the behavior of the COM server.

Instancing affects primarily out-of-process servers (any COM server in a separate executable file, rather than a DLL) and can assume three values:

Multiple Indicates that when several client applications require the COM object, the system starts multiple instances of the server

Single Indicates that, even when several client applications require the COM object, there is only one instance of the server application; it creates multiple internal objects to service the requests

Internal Indicates that the object can only be created inside the server; client applications cannot ask for one (this specific setting affects also in-process servers).

The second decision relates to the COM object's thread support, which is valid for in-process servers only (DLLs). The threading model is a joint decision of the client and the server application: If both sides agree on one model, it is used for the connection. If no agreement is found, COM can still set up a connection using marshaling, which can slow down the operations. Also keep in mind that a server must not only publish its threading model in the Registry (as a result of setting the option in the wizard); it must also follow the rules for that threading model in the code. Here are the key highlights of the various threading models:

Single Model No real support for threads. The requests reaching the COM server are serialized so the client can perform one operation at a time.

Apartment Model, or "Single-Threaded Apartment" Only the thread that created the object can call its methods. This means the requests for each server object are serialized, but other objects of the same server can receive requests at the same time. For this reason, the server object must take extra care to access only global data of the server (using critical sections, mutexes, or some other synchronization techniques). This is the threading model generally used for ActiveX controls inside Internet Explorer.

Free Model, or "Multithreaded Apartment" The client has no restrictions, which means multiple threads can use the same object at the same time. For this reason, every method of every object must protect itself and the nonlocal data it uses against multiple simultaneous calls. This threading model is more complex for a server to support than the Single and Apartment models, because even access to the object's own instance data must be handled with thread-safe care.

Both The server object supports both the Apartment model and the Free model.

Neutral Introduced in Windows 2000 and available only under COM+. This model indicates that multiple clients can call the object on different threads at the same time, but COM guarantees you that the same method is not invoked twice at the same time. Guarding for concurrent access to the object's data is required. Under COM, it is mapped to the Apartment model.

Initializing the COM Object

If you look at the definition of the `TComObject` class, you will notice it has a nonvirtual constructor. Actually it has multiple constructors, each calling the virtual `Initialize` method. For this reason, to set up a COM object properly, you should not define a new constructor (which will never be called), but instead override its `Initialize` method, as I've done in the `TNumber` class. Here is the final version of this class:

```
type
  TNumber = class(TComObject, INumber)
  private
    fValue: Integer;
  public
    function GetValue: Integer; virtual; stdcall;
    procedure SetValue (New: Integer); virtual; stdcall;
    procedure Increase; virtual; stdcall;
    procedure Initialize; override;
    destructor Destroy; override;
  end;
```

As you can see, I've also overridden the destructor of the class, because I wanted to test the automatic destruction of the COM objects provided by Delphi.

Testing the COM Server

Now that you've finished writing the COM server object, you can register and use it. Compile its code and then use the Run ➢ Register ActiveX Server menu command in Delphi. You do this to register the server on your own machine, updating the local Registry.

When you distribute this server, you should install it on the client computers. To accomplish this, you could write a REG file to install the server in the Registry. However, this is not the best approach, because the server already includes a function you can activate to register the server. This function can be activated by the Delphi environment, as you've seen, or in a few other ways:

- You can pass the COM server DLL as a command-line parameter to Microsoft's `RegSvr32.exe` program, found in the `\Windows\System` directory.
- You can use the similar `TRegSvr.exe` demo program that ships with Delphi. (The compiled version is in the `\Bin` directory, and its source code is in the `\Demos\ActiveX` directory.)
- You can let an installation builder program call the server's registration function.

Having registered the server, you can now turn to the client side of the example. This time, the example is called TestCom, and is stored in a separate directory. The program loads the server DLL through the COM mechanism, thanks to the server information present in the Registry, so it's not necessary for the client to know which directory the server resides in.

The form displayed by this program is similar to the one you used to test some of the DLLs in Chapter 10, "Libraries and Packages." In the client program, you must include the source code file with the interface and redeclare the COM server GUID. The program starts with all the buttons disabled (at design time), and it enables them only after an object has been created. This way, if an exception is raised while creating one of the objects, the buttons related to the object won't be enabled:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // create first object
  Num1 := CreateComObject (Class_Number) as INumber;
  Num1.SetValue (SpinEdit1.Value);
  Label1.Caption := 'Num1:' + IntToStr (Num1.GetValue);
  Button1.Enabled := True;
  Button2.Enabled := True;

  // create second object
  Num2 := CreateComObject (Class_Number) as INumber;
  Label2.Caption := 'Num2: ' + IntToStr (Num2.GetValue);
  Button3.Enabled := True;
  Button4.Enabled := True;
end;
```

Notice in particular the call to `CreateComObject` and the following `as` cast. The API call starts the COM object-construction mechanism I've already described in detail. This call also dynamically loads the server DLL. The return value is an `IUnknown` object. This object must be converted to the proper interface type before it is assigned to the `Num1` and `Num2` fields, which now have the interface type `INumber` as their data type.

WARNING *To downcast an interface to the type,* always *use the* `as` *cast, which for interfaces performs a* `QueryInterface` *call behind the scenes. Alternatively, you can do a direct* `QueryInterface` *or* `Supports` *call. In the case of interfaces, the* `as` *cast (or a specific function call) is the only way to* extract *an interface from another interface. Casting an interface pointer to another interface pointer directly is an error—never do it.*

The program also has a button (toward the bottom of the form) with an event handler that creates a new COM object used to get the value of the number following 100. To see why I added this method to the example, click the button in the message showing the result. You'll see a second message indicating that the object has been destroyed. This demonstrates that letting an interface variable go out of scope invokes the object's `_Release` method, decreases the object's reference count, and destroys the object if its reference count reaches zero.

The same thing happens to the other two objects as soon as the program terminates. Even if the program doesn't explicitly do so, the two objects are indeed destroyed, as the message shown by their `Destroy` destructor clearly demonstrates. This happens because they were declared to be of an interface type, and Delphi will use reference counting for them. By the way, in case you want to destroy a COM object reference with an interface, you cannot call a `Free` method (interfaces don't have `Free`) but can assign `nil` to the interface variable; this causes the removal of the reference and possibly the destruction of the object.

Using Interface Properties

As a further small step, you can extend the example by adding a property to the `INumber` interface. When you add a property to an interface, you indicate the data type and then the `read` and `write` directives. You can have read-only or write-only properties, but the `read` and `write` clauses must always refer to a method because interfaces don't hold anything but methods.

Here is the updated interface, which is part of the PropCom example:

```
type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer read GetValue write SetValue;
    procedure Increase; stdcall;
  end;
```

I've given this interface a new name and, even more important, a new interface ID. I could have inherited the new interface type from the previous one, but doing so would have provided no real advantage. COM by itself doesn't support inheritance, and from the perspective of COM, all interfaces are different because they have different interface IDs. Needless to say, in Delphi you can use inheritance to improve the structure of the code of the interfaces and of the server objects implementing them.

In the PropCom example, I've updated the server class declaration by referring to the new interface and providing a new server object ID. The client program (called TestProp) can now use the `Value` property instead of the `SetValue` and `GetValue` methods. Here is a small excerpt from the `FormCreate` method:

```
Num1 := CreateComObject (Class_NumPropServer) as INumberProp;
Num1.Value := SpinEdit1.Value;
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

The difference between using methods and properties for an interface is only syntactical, because interface properties cannot access private data as Delphi class properties can. By using properties, you can make the code a little more readable.

Calling Virtual Methods

You've built a couple of examples based on COM, but you might still feel uncomfortable with the idea of a program calling methods of objects that are created within a DLL. How is this possible if those methods are not exported by the DLL? The COM server (the DLL) creates an object and returns it to the calling application. By doing this, the DLL creates an object with a *virtual method table* (VMT). To be more precise, the object has a VMT for its class plus virtual method tables for each of the interfaces it implements.

The main program receives back an interface variable with the virtual method table of the requested interface. This VMT can be used to invoke methods, but also can be used to query for other interfaces supported by the COM object (since the `QueryInterface` method is available as part of the `IUnknown` interface VMT).

The main program doesn't need to know the memory address of those methods, because the objects know it, exactly as they do with a polymorphic call. But COM is even more powerful than this: You don't have to know which programming language was used to create the object, provided its VMT follows the standard dictated by COM.

TIP *The COM-compatible VMT implies a strange effect. The method names are not important, provided their address is in the proper position in the VMT. This is why you can map a method of an interface to a function implementing it.*

To sum things up, COM provides a language-independent binary standard for objects. The objects you share among modules are compiled, and their VMT has a particular structure determined by COM and not by the development environment you've used.

Automation

Up to now, you have seen that you can use COM to let an executable file and a library share objects. Most of the time, however, users want applications that can talk to each other. One of the approaches you can use for this goal is Automation (previously called *OLE Automation*). After presenting a couple of examples that use custom interfaces based on type libraries, I'll cover the development of Word and Excel Automation controllers, showing how to transfer database information to those applications.

NOTE *The current Microsoft documentation uses the term* Automation *instead of* OLE Automation, *and uses the terms* active document *and* compound document *instead of* OLE document. *This book tends to use the new terminology, although the older "OLE" terminology is still indicated as it was probably more clear.*

In Windows, applications don't exist in separate worlds; users often want them to interact. The Clipboard and DDE offer a simple way for applications to interact, as users can copy and paste data between applications. However, more and more programs offer an Automation interface to let other programs drive them. Beyond the obvious advantage of programmed automation compared to manual user operations, these interfaces are completely language-neutral, so you can use Delphi, C++, Visual Basic, or a macro language to drive an Automation server regardless of the programming language used to write it. Automation is straightforward to implement in Delphi, thanks to the extensive work by the compiler and VCL to shield developers from its intricacies. To support Automation, Delphi provides a wizard and a powerful type-library editor, and it supports dual interfaces. When you use an in-process DLL, the client application can use the server and call its methods directly, because they are in the same address space. When you use Automation, the situation is more complex. The client (called the *controller*) and the server are generally two separate applications running in different address spaces. For this reason, the system must dispatch the method calls using a complex parameter passing mechanism called *marshaling* (something I won't cover in detail).

Technically, supporting Automation in COM implies implementing the `IDispatch` interface, declared in Delphi in the System unit as:

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall;
  end;
```

The first two methods return type information; the last two are used to invoke an actual method of the Automation server. Actually the invocation is performed only by the last method, `Invoke`, while `GetIDsOfNames` is used to determine the dispatch id (required by `Invoke`) from the method name. When you create an Automation server in Delphi all you have to do is define a type library and implement its interface. Delphi provides everything else through compiler magic and VCL code (actually a portion of the VCL originally called DAX framework).

The role of `IDispatch` becomes more obvious when you consider that there are three ways a controller can call the methods exposed by an Automation server:

- It can ask for the execution of a method, passing its name in a string, in a way similar to the dynamic call to a DLL. This is what Delphi does when you use a variant (see the following note) to call the Automation server. This technique is easy to use, but it is rather slow and provides little compiler type-checking. It implies a call to `GetIDsOfNames` followed by one to `Invoke`.
- It can import the definition of a Delphi dispatch interface (`dispinterface`) for the object on the server and call its methods in a more direct way (dispatching a number, that is calling `Invoke` directly as the `DispId` of each method is known at compile time). This technique is based on interfaces and allows the compiler to check the types of the parameters and produces faster code, but it requires a little more effort from the programmer (namely the use of a type library). Also, you end up binding your controller application to a specific version of the server.
- It can call the interface directly, through the interface *vtable*, i.e. treating it as a normal COM object. This works in most cases as most Automation servers interfaces provide dual interfaces (that is support both `IDispatch` and a plain COM interface).

In the following examples, you'll use these techniques and compare them a little further.

NOTE *You can use a variant to store a reference to an Automation object. In the Delphi language a variant is a* type-variant *data type, that is a variable that can assume different data types as its value. Variant data types include the basic ones (such as Integers, strings, characters, and Boolean values) but also the* `IDispatch` *interface type. Variants are type-checked at run time; this is why the compiler can compile the code even if it doesn't know about the methods of the Automation server, as we'll see later on.*

Dispatching an Automation Call

The most important difference between the two approaches is that the second generally requires a *type library*, one of the foundations of COM. A type library is basically a collection of type information, which is often found also in a COM object (with not dispatch support). This collection generally describes all the elements (objects, interfaces, and other type information) made available by a generic COM server oran Automation server. The key difference between a type library and other descriptions of these elements (such as C or Pascal code) is that a type library is language-independent. The type elements are defined by COM as a subset of the standard elements of programming languages, and any development tool can use them. Why do you need this information?

I've mentioned earlier that if you invoke a method of an Automation object using a variant, the Delphi compiler doesn't need to know about this method at compile time. A small code fragment using Word's old Automation interface, registered as `Word.Basic`, illustrates how simple it is for a programmer:

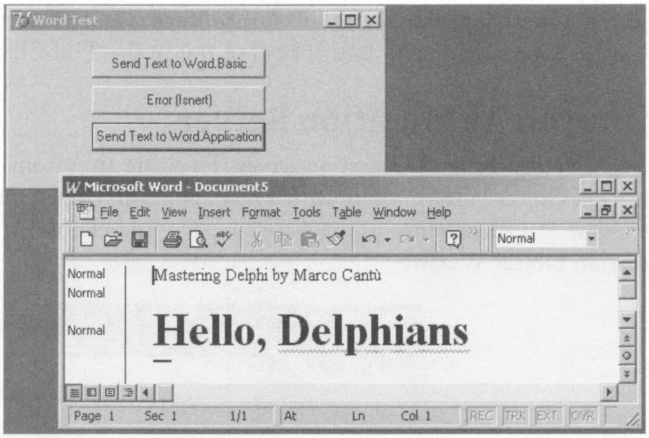
```
var
  VarW: Variant;
```

```
begin
  VarW := CreateOleObject ('Word.Basic');
  VarW.FileNew;
  VarW.Insert ('Mastering Delphi by Marco Cantù');
```

NOTE *As you'll see later, recent versions of Word still register the* `Word.Basic` *interface, which corresponds to the internal WordBasic macro language, but they also register the new interface* `Word.Application`*, which corresponds to the VBA macro language. Delphi provides components that simplify the connection with Microsoft Office applications, introduced later on in this chapter.*

These three lines of code start Word (unless it was already running), create a new document, and add a few words to it. You can see the effect of this application in Figure 12.3.

FIGURE 12.3
The Word document is being created and composed by the WordTest Delphi application.



Word Test
Send Text to Word.Basic
Error (Isnert)
Send Text to Word.Application
Microsoft Word - Document5
File Edit View Insert Format Tools Table Window Help
Normal
Normal
Normal
Normal
Page 1 Sec 1 1/1 At Ln Col 1 REC TRK EXT OVR

Mastering Delphi by Marco Cantù

Hello, Delphians

Unfortunately, the Delphi compiler has no way to check whether the methods exist. Doing all the type checks at run time is risky, because if you make even a minor spelling error in a function name, you get no warning about your error until you run the program and reach that line of code. For example, if you type **`VarW.Isnert`**, the compiler will not complain about the misspelling, but at run time, you'll get an error. Because it doesn't recognize the name, Word assumes the method does not exist.

Although the `IDispatch` interface supports the approach you've just seen, it is also possible—and safer—for a server to export the description of its interfaces and objects using a type library. This type library can then be converted by a specific tool (such as Delphi) into definitions written in the language you want to use to write your client or controller program (such as the Delphi language). This makes it possible for a compiler to check whether the code is correct and for you to use Code Completion and Code Parameters in the Delphi editor.

Once the compiler has done its checks, it can use either of two different techniques to send the request to the server. It can use a plain VTable (that is, an entry in an interface type declaration), or it can use a `dispinterface` (dispatch interface). You used an interface type declaration earlier in this chapter, so it should be familiar. A `dispinterface` is basically a way to map each entry in an interface to a number. Calls to the server can then be dispatched by number calling `IDipatch.Invoke` only, without the extra step of calling `IDispatch.GetIDsOfNames`. You can consider this an intermediate technique, in between dispatching by function name and using a direct call in the VTable.

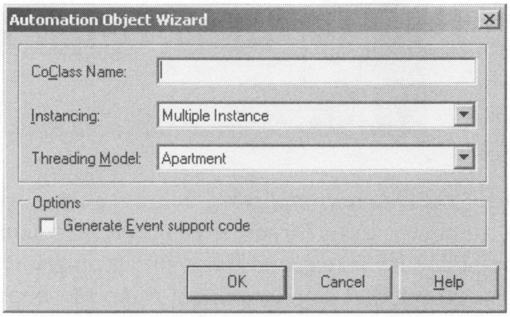
NOTE *The term* dispinterface *is a keyword. A* `dispinterface` *is automatically generated by the type-library editor for every interface. Along with* `dispinterface`*, Delphi uses other related keywords:* `dispid` *indicates the number to associate with each element;* `readonly` *and* `writeonly` *are optional specifiers for properties.*

The term used to describe this ability to connect to a server in two different ways, using a more dynamic or a more static approach, is *dual interfaces*. When writing a COM controller, you can choose to access the methods of a server two ways: you can use late binding and the mechanism provided by the `dispinterface`, or you can use early binding and the mechanism based on the VTables, the interface types.

It is important to keep in mind that (along with other considerations) different techniques result in faster or slower execution. Looking up a function by name (and doing the type checking at run time) is the slowest approach, using a `dispinterface` is much faster, and using the direct VTable call is the fastest approach. You'll do this kind of test in the TLibCli example, later in this chapter.

Writing an Automation Server

Let's begin by writing an Automation server. To create an Automation object, you can use Delphi's Automation Object Wizard. Begin with a new application, open the Object Repository by selecting File ➢ New ➢ Other, move to the ActiveX page, and choose Automation Object. You'll see the Automation Object Wizard:


Automation Object Wizard

CoClass Name:

Instancing: Multiple Instance

Threading Model: Apartment

Options

☐ Generate Event support code

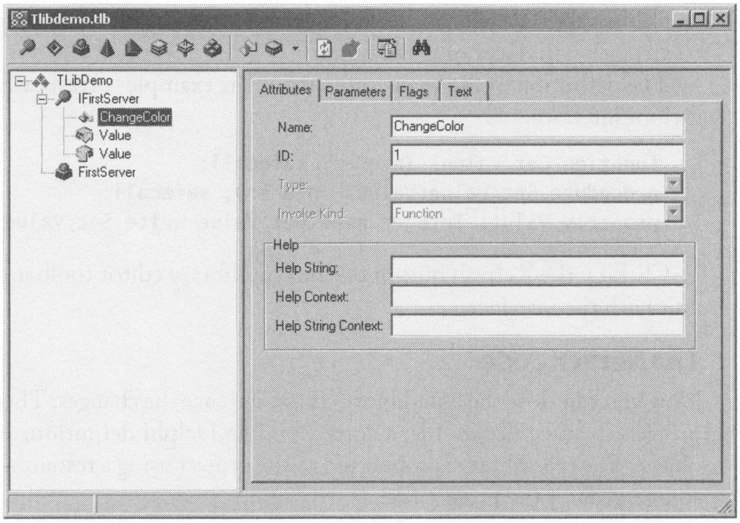
In this wizard, enter the name of the class (without the initial *T*, because it will be added automatically for you to the Delphi implementing class) and click OK. Delphi will now open the type-library editor.

TIP *Delphi can generate Automation servers that also export events. Select the corresponding check box in the Wizard, and Delphi will add the proper entries in the type library and in the source code it generates.*

The Type-Library Editor

You can use the type-library editor to define a type library in Delphi. Figure 12.4 shows its window after I've added some elements to it. The type-library editor allows you to add methods and properties to the Automation server object you've just created or to a COM object that was created using the COM Object wizard. Once you do, it can generate both the type library (TLB) file and the corresponding Delphi language source code stored in a unit called a type library import unit.

FIGURE 12.4
The type-library editor, showing the details of an interface



Name: ChangeColor
ID: 1
Type:
Invoke Kind: Function
Help String:
Help Context:
Help String Context:

I have two relevant suggestions to let you work better with Delphi's type-library editor. The first and most simple is that if you right-click on the toolbar and turn on the Text Labels option you'll see in each toolbar button a caption with its effect, making the editor easier to use. The second more relevant suggestion is to go to the Type Library page of Delphi's Environment Options dialog box and choose the Pascal language radio button over the IDL language radio button. This setting determines the notation used by the type-library editor to display methods and parameters, and even to edit the types of the parameters of a method or the type of a property. Unless you are used to writing COM code in C or C++, you'd probably prefer thinking in terms of Delphi rather than in terms of IDL.

WARNING *In this portion of the book I'll describe how to interact with the type-library editor when you have this setting, since also providing a description in terms of IDL would be both needlessly confusing and rather complex.*

To build a first example, you can add a property and a method to the server by using the editor's corresponding toolbar buttons and typing their names either in the Tree View control on the left side of the window or in the Name edit box on the right side. You add these two elements to an interface, which I've called `IFirstServer`.

For the procedure you'll be able to define the parameters in the Parameters page, for a function you'd also be able to set a return type in the same page. In this specific case the `ChangeColor` method has no parameters and its Delphi definition would be:

```
procedure ChangeColor; safecall;
```

NOTE *The methods contained in Automation interfaces in Delphi generally use the* `safecall` *calling convention. It wraps a* `try/except` *block around each method and provides a default return value indicating error or success. It also sets up a COM rich error object containing the exception message, so interested clients (such as Delphi clients) can re-create the server exception on the client side.*

Now you can add a property to the interface by clicking the Property button on the type-library editor's toolbar. Again, you can type a name for it, such as `Value`, and select a data type in the Type combo box. Besides selecting one of the many types already listed, you can also enter other types directly, particularly interfaces of other objects.

The definition of the `Value` property of the example corresponds to the following elements of the Delphi interface:

```
function Get_Value: Integer; safecall;
procedure Set_Value(Value: Integer); safecall;
property Value: Integer read Get_Value write Set_Value;
```

Clicking the Refresh button on the type-library editor toolbar generates (or updates) the Delphi unit with the interface.

The Server Code

Now you can close the type-library editor and save the changes. This operation adds three items to the project: the type library file, a corresponding Delphi definition, and the declaration of the server object. The type library is connected to the project using a resource-inclusion statement, added to the source code of the project file:

```
{$R *.TLB}
```

You can always reopen the type-library editor by using the View ➢ Type Library command or by selecting the proper TLB file in Delphi's normal File Open dialog box.

As mentioned earlier, the type library is also converted into an interface definition and added to a new Delphi unit. This unit is quite long, so I've listed in the book only its key elements. The most important part is the new interface declaration:

```
type
  IFirstServer = interface(IDispatch)
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; safecall;
    function Get_Value: Integer; safecall;
    procedure Set_Value(Value: Integer); safecall;
    property Value: Integer read Get_Value write Set_Value;
  end;
```

Then comes the `dispinterface`, which associates a number with each element of the `IFirstServer` interface:

```
type
  IFirstServerDisp = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
  end;
```

The last portion of the file includes a creator class, which is used to create an object on the server (and for this reason used on the client side of the application, not on the server side):

```
type
  CoFirstServer = class
```

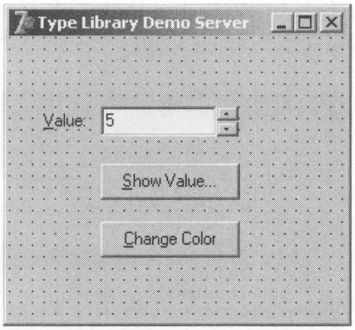
```
    class function Create: IFirstServer;
    class function CreateRemote(const MachineName: string): IFirstServer;
  end;
```

All the declarations in this file (I've skipped some others) can be considered internal, hidden implementation support. You don't need to understand them fully in order to write most Automation applications.

Finally, Delphi generates a file containing the implementation of your Automation object. This unit is added to the application and is the one you'll work on to finish the program. This unit declares the class of the server object, which must implement the interface you've just defined:

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
  end;
```

Delphi already provides the skeleton code for the methods, so you only need to complete the lines in between. In this case, the three methods refer to a property and two methods I've added to the form. In general, you should not add code related to the user interface inside the class of the server object. I've done it because I wanted to be able to change the `Value` property and have a visible side effect (displaying the value in an edit box). Here you can see this form at design time:



Registering the Automation Server

The unit containing the server object has one more statement, added by Delphi to the `initialization` section:

```
initialization
  TAutoObjectFactory.Create(ComServer, TFirstServer, Class_FirstServer,
    ciMultiInstance);
end.
```

NOTE *In this case, I've selected multiple instancing. For the various instancing styles possible in COM, see the sidebar "COM Instancing and Threading Models" earlier in this chapter.*

This is not very different from the creation of class factories you saw at the beginning of this chapter. The ComServer unit hooks the `InitProc` system function to register all COM objects as part of the COM server application startup. The execution of this code is triggered by the `Application.Initialize` call, which Delphi adds by default to the project source code of any program.

You can add the server information to the Windows Registry by running this application on the target machine (the computer where you want to install the Automation server), or by running it and passing to it the `/regserver` parameter on the command line. You can do this by selecting Start ➢ Run, by creating a shortcut in Explorer, or by running the program within Delphi after you've entered a command-line parameter (using the Run ➢ Parameters command). Another command-line parameter, `/unregserver`, is used to remove this server from the Registry.

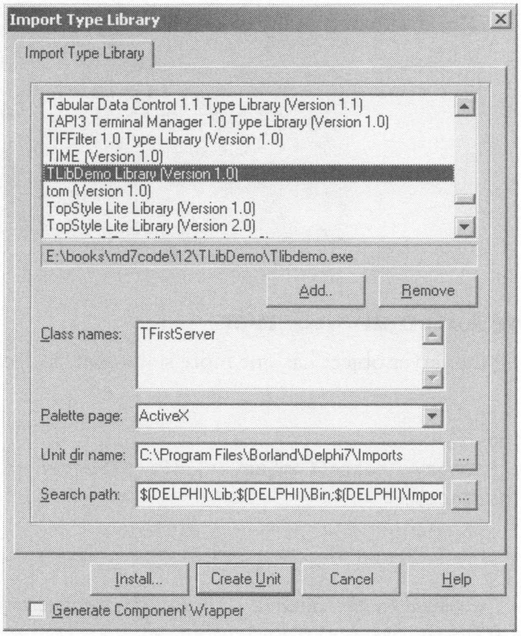
Writing a Client for the Server

Now that you have built a server, you can prepare a client program to test it. This client can connect to the server either by using variants or by using the new type library. This second approach can be implemented manually or by using Delphi techniques for wrapping components around Automation servers. You'll try all these approaches.

Create a new application—I've called it TLibCli—and import the server's type library, using the Project ➢ Import type library menu command of the Delphi IDE. This command shows the Import Type Library dialog box, visible in Figure 12.5. This dialog lists registered COM servers having a type library in the upper portion. You can add other projects to this list pressing the Add button and browsing for the proper file module. The lower portion of the Import Type Library dialog box shows some details of the selected library (such as the list of server objects) and about the type library import unit this dialog box is going to produce as you press the Create Unit button (or the Install button).

FIGURE 12.5

Delphi's Type Library Import dialog box.



Tabular Data Control 1.1 Type Library (Version 1.1)
TAPI3 Terminal Manager 1.0 Type Library (Version 1.0)
TIFFilter 1.0 Type Library (Version 1.0)
TIME (Version 1.0)
TLibDemo Library (Version 1.0)
tom (Version 1.0)
TopStyle Lite Library (Version 1.0)
TopStyle Lite Library (Version 2.0)

E:\books\md7code\12\TLibDemo\Tlibdemo.exe

Class names: TFirstServer

Palette page: ActiveX

Unit dir name: C:\Program Files\Borland\Delphi7\Imports

Search path: $(DELPHI)\Lib;$(DELPHI)\Bin;$(DELPHI)\Impor

☐ Generate Component Wrapper

WARNING *Do not add the type library to the client application, because you are writing the Automation controller, not a server. A controller's Delphi project should not include the type library of the server it connects to.*

The type library import unit is named by Delphi after the type library, with an `_TLB` at the end. In this case the unit name is `TlibdemoLib_TLB`. I've already mentioned that one of the elements of this unit, generated also by the type library editor, is the *creation* class. I've shown you the interface of this class, but here is the implementation of the first of the two functions:

```
class function CoFirstServer.Create: IFirstServer;
begin
  Result := CreateComObject(Class_FirstServer) as IFirstServer;
end;
```

You can use it to create a server object (and possibly start the server application) on the same computer. As you can see in the code, the function is a shortcut for the `CreateComObject` call, which allows you to create an instance of a COM object if you know its GUID. As an alternative, you can use the `CreateOleObject` function, which requires as a parameter a ProgID, which is the registered name of the server. There is another difference between these two creation functions: `CreateComObject` returns an object of the `IUnknown` type, whereas `CreateOleObject` returns an object of the `IDispatch` type.

In this example, let's use the `CoFirstServer.Create` shorthand. When you create the server object, you get as a return value an `IFirstServer` interface. You can use it directly or store it in a variant variable. Here is an example of the first approach:

```
var
  MyServer: Variant;
begin
  MyServer := CoFirstServer.Create;
  MyServer.ChangeColor;
```

This code, based on variants, is not very different from that of the first controller you built in this chapter (the one that used Microsoft Word). Here is the alternate code, which has the same effect:

```
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

You've already seen how you can use the interface and the variant. What about the dispatch interface? You can declare a variable of the dispatch interface type, in this case:

```
var
  DMyServer: IFirstServerDisp;
```

Then you can use it to call the methods as usual, after you've assigned an object to it by casting the object returned by the creator class:

```
DMyServer := CoFirstServer.Create as IFirstServerDisp;
```

INTERFACES, VARIANTS, AND DISPATCH INTERFACES: TESTING THE SPEED DIFFERENCE

As I mentioned in the section introducing type libraries, one of the differences between these approaches is speed. It is complicated to assess the exact performance of each technique because many factors are involved. I've added a simple test to the TLibCli example among the demos for this chapter, to give you an idea. The code for the test is a loop that accesses the `Value` of the server 100 times. The output of the program shows the timing, which is determined by calling the `GetTickCount` API function before and after executing the loop. (Two alternatives are to use Delphi's own time functions, which are slightly less precise, or to use the very precise timing functions of the multimedia support unit, MMSystem.)

With this program, you can roughly compare the output obtained by calling this method based on an interface, the corresponding version based on a variant, and a third version based on a dispatch interface. Looking at the timing of the example, you should see that interfaces are quicker and variants are slower, with dispatch interfaces falling in between but closer to interfaces.

The Scope of Automation Objects

Another important element to keep in mind is the *scope* of the Automation objects. Variants and interface objects use reference-counting techniques, so if a variable that is related to an interface object is declared locally in a method, then at the end of the method the object will be destroyed and the server may terminate (if all the objects created by the server have been destroyed). For example, writing a method with this code produces minimal effect:

```
procedure TClientForm.ChangeColor;
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
end;
```

Unless the server is already active, a copy of the program is created and the color is changed, but then the server is immediately closed as the interface-typed object goes out of scope. The alternative approach I've used in the TLibCli example declares the object as a field of the form and creates the COM objects at startup, as in this procedure:

```
procedure TClientForm.FormCreate(Sender: TObject);
begin
  IMyServer := CoFirstServer.Create;
end;
```

With this code, as the client program starts, the server program is immediately activated. At program termination, the form field is destroyed and the server closes. A further alternative is to declare the object in the form, but then create it only when it is used, as in these two code fragments:

```
// MyServerBis: Variant;
if varType (MyServerBis) = varEmpty then
  MyServerBis := CoFirstServer.Create;
MyServerBis.ChangeColor;
```

```
// IMyServerBis: IFirstServer;
if not Assigned (IMyServerBis) then
  IMyServerBis := CoFirstServer.Create;
IMyServerBis.ChangeColor;
```

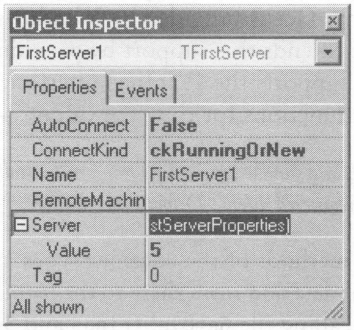
NOTE *A variant is initialized to the* `varEmpty` *type when it is created. If you instead assign the value null to the variant, its type becomes* `varNull`*. Both* `varEmpty` *and* `varNull` *represent variants with no value assigned, but they behave differently in expression evaluation. The* `varNull` *value always propagates through an expression (making it a null expression), whereas the* `varEmpty` *value quietly disappears.*

The Server in a Component

When creating a client program for your server or any other Automation server, you can use a better approach: wrapping a Delphi component around the COM server. If you look at the final portion of the `TlibdemoLib_TLB` file, you can find the declaration of a `TFirstServer` class inheriting from `TOleServer`. This is a component generated when importing the library, which the system registers in the unit's `Register` procedure.

If you add this unit to a package, the new server component will become available on the Delphi Component Palette (in the ActiveX page, by default). The generation of the code of this component is controlled by a check box at the bottom of the Import Type Library dialog box, already shown in Figure 12.5.

I've created a new package, PackAuto, available in a directory having the same name. In this package, I added the directive `LIVE_SERVER_AT_DESIGN_TIME` in the Directories/Conditionals page of the package's Project Options dialog box. This directive enables an extra feature that you don't get by default: At design time, the server component will have an extra property that lists as subitems all the properties of the Automation server:



FirstServer1 TFirstServer

AutoConnect	**False**
ConnectKind	**ckRunningOrNew**
Name	FirstServer1
RemoteMachin	
⊟ Server	stServerProperties)
Value	**5**
Tag	0

WARNING *The* `LIVE_SERVER_AT_DESIGN_TIME` *directive should be used with care with the most complex Automation servers (including programs such as Word, Excel, PowerPoint, and Visio). Certain servers must be in a particular mode before you can use some properties of their automation interfaces. Because this feature is problematic at design time for many servers, it is not active by default.*

As you can see in the Object Inspector, the component has few properties. `AutoConnect` indicates when to activate the COM server. When the value is True the server object is loaded as soon as the wrapper component is created (both at runtime and design time). When the `AutoConnect` property is set to False, the Automation server is loaded only the first time one of its methods is called. Another property, `ConnectKind`, indicates how to establish the connection with the server. It can always start a new instance (ckNewInstance), use the running instance (ckRunningInstance, which shows an error message if the server is not already running), or select the current instance or start a new one if none is available (ckRunningOrNew). Finally, you can ask for a remote server with ckRemote and directly attach a server in the code after a manual connection with ckAttachToInterface.

NOTE *To connect to an existing object, this needs to be registered in the Running Object Table (ROT). The registration must be performed by the server calling the* `RegisterActiveObject` *API function. Of course, only one instance for each COM server can be registered at a given time.*

COM Data Types

COM dispatching doesn't support all the data types available in Delphi. This is particularly important for Automation, because the client and the server are often executed in different address spaces, and the system must move (or *marshal*) the data from one side to the other. Also keep in mind that COM interfaces should be accessible by programs written in any language.

COM data types include basic data types such as Integer, SmallInt, Byte, Single, Double, WideString, Variant, and WordBool (but not Boolean).

In addition to the basic data types, you can use COM types for complex elements such as fonts, string lists, and bitmaps, using the `IFontDisp`, `IStrings`, and `IPictureDisp` interfaces. The following sections describe the details of a server that provides a list of strings and a font to a client.

EXPOSING STRINGS LISTS AND FONTS

The ListServ example is a practical demonstration of how you can expose two complex types, such as a list of strings and a font, from an Automation server written in Delphi. I've chosen these two specific types because they are both supported by Delphi.

The `IFontDisp` interface is provided by Windows and is available in the ActiveX unit. The AxCtrls Delphi unit extends this support by providing conversion methods like `GetOleFont` and `SetOleFont`. Delphi supports the `IStrings` interface in the StdVCL unit, and the AxCtrls unit provides conversion functions for this type (along with a third type I won't use here, `TPicture`).

WARNING *To run this and similar applications, you must install and register the StdVCL library on the client computer. On your computer, it is registered during Delphi's installation.*

The `Set` and `Get` methods of the complex types' properties copy information from the COM interfaces to the local data and from there to the form, and vice versa. The strings' two methods, for example, do this by calling the `GetOleStrings` and `SetOleStrings` Delphi functions. The client application used to demonstrate this feature is called ListCli. The two programs are complex; but rather than list their details here I've decided to leave the source code for your study, because Delphi programmers seldom use this advanced technique.

Using Office Programs

So far, you've built both the client and the server side of the Automation connection. If your aim is just to let two applications you've built cooperate, this is a useful technique, although it is not the only one. You've seen the use of memory-mapped files in Chapter 10. (Another technique not covered in this edition of the book is the use of the `wm_CopyData` message.) The real value of Automation is that it is a standard, so you can use it to integrate your Delphi programs with other applications your users own. A typical example is the integration of a program with Office applications, such as Microsoft Word and Microsoft Excel, or even with stand-alone applications, such as AutoCAD.

Integration with these applications provides a two-fold advantage:

- You can let your users work in an environment they know—for example, generating reports and memos from database data in a format they can easily manipulate.
- You can avoid implementing complex functionality from scratch, such as writing your own word-processing code inside a program. Instead of just reusing components, you can reuse complex applications.

This approach also has some drawbacks, which are worth mentioning:

- The user must own the application you plan to integrate with, and they may also need a recent version of it to support all the features you are using in your program.
- You have to learn a new programming architecture, often with limited documentation at hand. It is true that you are still using the Delphi language, but the code you write depends on the data types, the types introduced by the server, and in particular a collection of interrelated classes that are often difficult to understand.
- You might end up with a program that works only with a specific version of the server application, especially if you try to optimize the calls by using interfaces instead of variants. In particular, Microsoft does not attempt to maintain script compatibility between major releases of Word or other Office applications.

Delphi simplifies the use of Microsoft Office applications by preinstalling some ready-to-use components that wrap the Automation interface of these servers. These components, available in the Servers page of the Palette, are installed using the same technique I demonstrated in the last section. The real plus lies in the technique of creating components to wrap existing Automation servers, rather than in the availability of predefined server components. Notice also that these Office components exist in different versions depending on your version of the Microsoft suite: All components are installed, but only one set is registered at design time, according to your choice in the Delphi install program. You can change this setting later by removing the related component package and adding a new one.

You won't see an actual example in this section because it is very difficult to write a program that works with all of the different versions of Microsoft Office. You'll find same sample code and tips in Essential Delphi (see Appendix C for instructions on how to download this free ebook).

Using Compound Documents

Compound documents is Microsoft's name for the technology that allows in-place editing of a document within another document (for example, a picture in a Word document). This technology originated the term *OLE*, but its role is now definitely more limited than Microsoft envisioned when it was introduced in the early 1990s. Compound documents have two capabilities: *object linking* and *embedding* (hence the term *OLE*):

- *Embedding* an object in a compound document corresponds to a smart version of the copy and paste operations you perform with the Clipboard. The key difference is that when you copy an OLE object from a server application and paste it into a container application, you copy both the data and some information about the server (its GUID). This allows you to activate the server application from within the container to edit the data.
- *Linking* an object to a compound document copies only a reference to the data and the information about the server. You generally activate object linking by using the Clipboard and performing a Paste Link operation. When editing the data in the container application, you modify the original data, which is stored in a separate file.

Because the server program refers to an entire file (only part of which may be linked in the client document), the server will be activated in a stand-alone window, and it will act upon the entire original file, not just the data you've copied. When you have an embedded object, however, the container may support visual (or *in-place*) editing, which means you can modify the object in context inside the container's main window. The server and container application windows, their menus, and their toolbars are merged automatically, allowing the user to work in a single window on several different object types—and therefore with several different OLE servers—without leaving the window of the container application.

Another key difference between embedding and linking is that an embedded object's data is stored and managed by the container application. The container saves the embedded object in its own files. By contrast, a linked object physically resides in a separate file, which is handled by the server exclusively, even if the link refers only to a small portion of the file. In both cases, the container application doesn't have to know how to handle the object and its data—not even how to display it—without the help of the server. Considering the relative slowness of OLE and the amount of work necessary to develop COM servers, you can understand why this approach never took off.

Compound document containers can support COM in varying degrees. You can place an object in a container by inserting a new object, by pasting or *paste-linking* one from the Clipboard, by dragging one from another application, and so on. Once the object is placed in the container, you can then perform operations on it, using the server's available *verbs*, or actions. Usually the *Edit verb* is the default action—the action performed when you double-click on the object. For other objects, such as video or sound clips, *Play* is defined as the default action. You can typically see the list of actions supported by the current contained object by right-clicking it. The same information is available in many programs via the Edit ➢ Object menu item, which displays a submenu that lists the available verbs for the current object.

The Container Component

To create a COM container application in Delphi, place an OleContainer component in a form. Then select the component and right-click to activate its shortcut menu, which will include an Insert Object command. When you select this command, Delphi displays the standard OLE Insert Object dialog box. This dialog box allows you to choose from one of the server applications registered on the computer.

Once the COM object is inserted in the container, the control container component's shortcut menu will include several more custom menu items. The new menu items include commands to change the properties of the COM object, insert another object, copy the existing object, or remove the existing object. The list also includes the verbs (actions) of the object (such as Edit, Open, or Play). Once you have inserted a COM object in the container, the corresponding server will launch to let you edit the new object. As soon as you close the server application, Delphi updates the object in the container and displays it at design time in the form of the Delphi application you are developing.

If you look at the textual description of a form containing a component with an object inside, you'll notice a `Data` property, which contains the COM object's data. Although the client program stores the object's data, it doesn't know how to handle and show that data without the help of the proper server (which must be available on the computer where you run the program). This means the COM object is *embedded.*

To fully support compound documents, a program should provide a menu and a toolbar or panel. These extra components are important because in-place editing implies a merging of the client's user interface and the server program's user interface. When the COM object is activated in place, some of the pull-down menus on the server application's menu bar are added to the container application's menu bar.

Menu merging is handled almost automatically by Delphi. You only need to set the proper indexes for the menu items of the container, using the `GroupIndex` property. Any menu item with an odd index number is replaced by the corresponding element of the active OLE object. Specifically, the File (0) and Window (4) pull-down menus belong to the container application. The Edit (1), View (3), and Help (5) pull-down menus (or the groups of pull-down menus with those indexes) are taken by the COM server. A sixth group, Object (2), can be used by the container to display another pull-down menu between the Edit and View groups, when the COM object is active. The OleCont demo program I've written to demonstrate these features allows a user to create a new object by calling the `InsertObjectDialog` method of the `TOleContainer` class.

Once a new object has been created, you can execute its primary verb using the `DoVerb` method. The program also displays a small toolbar with some bitmap buttons. I placed some `TWinControl` components in the form to let the user select them and thus disable the OleContainer. To keep this toolbar/panel visible while in-place editing is occurring, you should set its `Locked` property to True. This setting forces the panel to remain present in the application and not be replaced by a toolbar of the server.

To show what happens when you don't use this approach, I've added to the program a second panel with more buttons. Because I haven't set its `Locked` property, this new toolbar will be replaced with that of the active server. When in-place editing launches a server application that displays a toolbar, that server's toolbar replaces the container's toolbar, as you can see in the lower part of Figure 12.6.

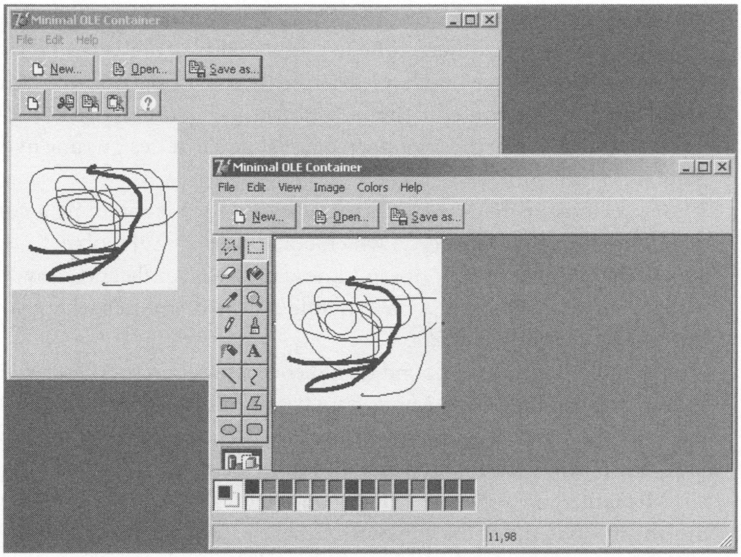
TIP *To make all the automatic resizing operations work smoothly, you should place the OLE container component in a panel component and align both of them to the client area of the form.*

Alternatively, you can create a COM object using the `PasteSpecialDialog` method, called in the example's `PasteSpecial1Click` event handler. Another standard COM dialog box, wrapped in a

Delphi function, shows the properties of the object; this dialog box is activated with the Object Properties item in the Edit pull-down menu by calling the `ObjectPropertiesDialog` method of the OleContainer component.

FIGURE 12.6

The second toolbar of the OleCont example (above) is replaced by the server's toolbar (below).



The last feature of the OleCont program is support for files. This is one of the simplest additions you can make, because the OLE container component already provides file support.

Using the Internal Object

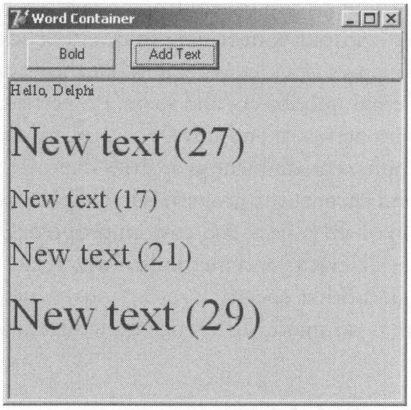
In the preceding program, the user determined the type of the internal object created by the program. In this case, there is little you can do to interact with the internal objects. Suppose, instead, that you want to embed a Word document in a Delphi application and then modify it by code. You can do this by using Automation with the embedded object, as demonstrated by the WordCont example (the name stands for *Word container*).

WARNING *Because the WordCont example includes an object of a specific type (a Microsoft Word document), it won't run if you don't have that application installed. Having a different version of the server might also create problems (I've tested the examples in this chapter only with Office 97). For other versions, you might have to rebuild the program following the same steps I used.*

In the example's form, I added an OleContainer component, set its `AutoActivate` property to aaManual (so that the only possible interaction is with my code), and added a toolbar with a couple of buttons. The code is straightforward, once you know that the embedded object corresponds to a Word document. Here is an example (you can see the effect of this code in Figure 12.7):

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Document, Paragraph: Variant;
```

```
begin
  // activate if not running
  if not (OleContainer1.State = osRunning) then
    OleContainer1.Run;
  // get the document
  Document := OleContainer1.OleObject;
  // add paragraphs, getting the last one
  Document.Paragraphs.Add;
  Paragraph := Document.Paragraphs.Add;
  // add text to the paragraph, using random font size
  Paragraph.Range.Font.Size := 10 + Random (20);
  Paragraph.Range.Text := 'New text (' +
    IntToStr (Paragraph.Range.Font.Size) + ')'#13;
end;
```



Hello, Delphi

New text (27)

New text (17)

New text (21)

New text (29)

FIGURE 12.7
The WordCont example shows how to use Automation with an embedded object.

Introducing ActiveX Controls

Microsoft's Visual Basic was the first program development environment to introduce the idea of supplying software components to the mass market, even if the concept of reusable software components is older than Visual Basic—it's well rooted in the theories of object-oriented programming (OOP). The first technical standard promoted by Visual Basic was *VBX*, a 16-bit specification that was fully available in Delphi 1. In moving to the 32-bit platforms, Microsoft replaced the VBX standard with the more powerful and more open *ActiveX* controls.

NOTE *ActiveX controls used to be called OLE controls (or OCX). The name change reflects a new marketing strategy from Microsoft rather than a technical innovation. Not surprisingly, then, ActiveX controls are usually saved in files with the* `.ocx` *extension.*

From a general perspective, an ActiveX control is not very different from a Windows control. The key difference is in the control's *interface*—the interaction between the control and the rest of the application. Typical Windows controls use a message-based interface; Automation objects and ActiveX controls use properties, methods, and events (like Delphi's own components).

Using COM jargon, an ActiveX control is a "compound document object which is implemented as an in-process server DLL and supports Automation, visual editing, and inside-out activation." Perfectly clear, right? Let's see what this definition means. COM servers can be implemented three ways:

- As stand-alone applications (for example, Microsoft Excel)
- As out-of-process servers—that is, executables files that cannot be run by themselves and can only be invoked by a server (for example, Microsoft Graph and similar applications)
- As in-process servers, such as DLLs loaded into the same memory space as the program using them

ActiveX controls can only be implemented using the last technique, which is also the fastest: as in-process servers. Furthermore, ActiveX controls are Automation servers. This means you can access properties of these objects and call their methods. You can see an ActiveX control in the application that is using it and interact with it directly in the container application window. This is the meaning of the term *visual editing*, or *in-place activation*. A single click activates the control rather than the double-click used by OLE documents, and the control is active whenever it is visible (which is what the term *inside-out activation* means) without your having to double-click it.

In an ActiveX control, properties can identify states, but they can also activate methods. Properties can refer to aggregate values, arrays, subobjects, and so on. Properties can also be dynamic (or read-only, to use the Delphi term). The properties of an ActiveX control are divided into groups: stock properties that most controls need to implement; ambient properties that offer information about the container (similar to the `ParentColor` and `ParentFont` properties in Delphi); extended properties managed by the container, such as the position of the object; and custom properties, which can be anything.

Events and methods are, well, events and methods. *Events* relate to a mouse click, a key press, the activation of a component, and other specific user actions. *Methods* are functions and procedures related to the control. There is no major difference between the ActiveX and Delphi concepts of events and methods.

ActiveX Controls Versus Delphi Components

Before I show you how to use and write ActiveX controls in Delphi, let's go over some of the differences between the two kinds of controls. ActiveX controls are DLL-based: When you use them, you need to distribute their code (the OCX file) along with the application using them. In Delphi, the components' code can be statically linked to the executable file or dynamically linked to it using a run-time package, so you can always choose whether to deploy a single large file or many smaller modules.

Having a separate file allows you to share code among different applications, as DLLs usually do. If two applications use the same control (or run-time package), you need only one copy of it on the hard disk and a single copy in memory. The drawback, however, is that if the two programs have to use two different versions (or builds) of the ActiveX control, some compatibility problems may arise. An advantage of having a self-contained executable file is that you will also have fewer installation problems.

The drawback of using Delphi components is not that there are fewer Delphi components than ActiveX controls, but that if you buy a Delphi component, you'll only be able to use it in Delphi and Borland C++Builder. If you buy an ActiveX control, on the other hand, you'll be able to use it in multiple development environments from multiple vendors. Even so, if you develop mainly in Delphi and find two similar components based on the two technologies, I suggest you buy the Delphi component—it will be more integrated with your environment, and therefore easier for you to use. Also, the native Delphi component will probably be better documented (from the Delphi perspective), and it will take advantage of Delphi and its language features not available in the general ActiveX interface, which is traditionally based on C and C++.

NOTE *In the .NET world, this situation will change completely. Not only will you be able to use any system component in a more seamless way, but you'll also be able to make Delphi components available to other .NET programming languages and tools.*

Using ActiveX Controls in Delphi

Delphi comes with some preinstalled ActiveX controls, and you can buy and install more third-party ActiveX controls. After this description of how ActiveX controls work in general, I'll demonstrate one in an example.

The Delphi installation process is simple:

1. Select Component ➢ Import ActiveX Control in the Delphi menu to open the Import ActiveX dialog box, where you can see the list of ActiveX control libraries registered in Windows.
2. Choose one, and Delphi will read its type library, list its controls, and suggest a filename for its unit.
3. If the information is correct, click the Create Unit button to view the Delphi language source code file created by the IDE as a *wrapper* for the ActiveX control.
4. Click the Install button to add this new unit to a Delphi package and to the Component Palette.

USING THE WEBBROWSER CONTROL

To build an example, I've used a preinstalled ActiveX control available in Delphi. Unlike the third-party controls, it is not available in the ActiveX page of the palette, but in the Internet page. The control, called WebBrowser, is a wrapper around Microsoft's Internet Explorer engine. The WebDemo example is a very limited web browser; it has a `TWebBrowser` ActiveX control covering its client area, a control bar at the top, and a status bar at the bottom. To move to a given web page, a user can type a URL in the toolbar's combo box, select one of the visited URLs (saved in the combo box), or click the Open File button to select a local file. Figure 12.8 shows an example of this program.

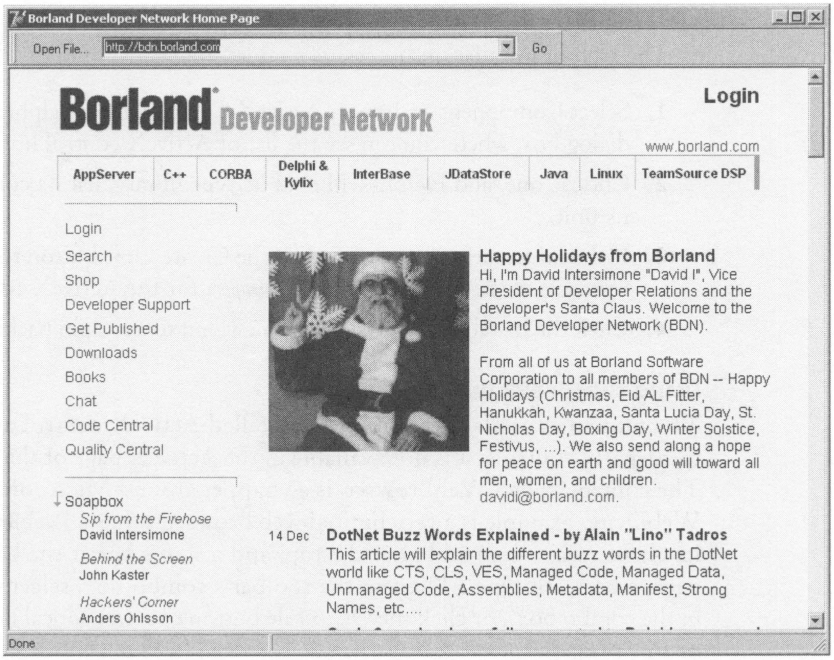
The implementation of the code used to select a web or local HTML file is in the `GotoPage` method:

```
procedure TForm1.GotoPage(ReqUrl: string);
begin
  WebBrowser1.Navigate (ReqUrl, EmptyParam, EmptyParam, EmptyParam,
    EmptyParam);
end;
```

`EmptyParam` is a predefined `OleVariant` you can use to pass a default value as a reference parameter. This is a handy shortcut you can use to avoid creating an empty `OleVariant` variable each time you need a similar parameter. The program calls the `GotoPage` method when the user presses the Open File button, or when the user presses the Enter key while in the combo box or clicks the Go button, as you can see in the source code. The program also handles four events of the WebBrowser control. When the download operations ends, the program updates the text of the status bar and also the combo box's drop-down list:

```
procedure TForm1.WebBrowser1DownloadComplete(Sender: TObject);
var
```

```
    NewUrl: string;
  begin
    StatusBar1.Panels[0].Text := 'Done';
    // add URL to combo box
    NewUrl := WebBrowser1.LocationURL;
    if (NewUrl <> '') and (ComboURL.Items.IndexOf (NewUrl) < 0) then
      ComboURL.Items.Add (NewUrl);
  end;
```


Borland Developer Network Home Page
Open File...
http://bdn.borland.com
Go
Borland Developer Network
Login
www.borland.com
AppServer
C++
CORBA
Delphi & Kylix
InterBase
JDataStore
Java
Linux
TeamSource DSP
Login
Search
Shop
Developer Support
Get Published
Downloads
Books
Chat
Code Central
Quality Central
Soapbox
Sip from the Firehose
David Intersimone
Behind the Screen
John Kaster
Hackers' Corner
Anders Ohlsson
14 Dec
Done

Happy Holidays from Borland
Hi, I'm David Intersimone "David I", Vice President of Developer Relations and the developer's Santa Claus. Welcome to the Borland Developer Network (BDN).

From all of us at Borland Software Corporation to all members of BDN -- Happy Holidays (Christmas, Eid AL Fitter, Hanukkah, Kwanzaa, Santa Lucia Day, St. Nicholas Day, Boxing Day, Winter Solstice, Festivus, ...). We also send along a hope for peace on earth and good will toward all men, women, and children.
davidi@borland.com.

DotNet Buzz Words Explained - by Alain "Lino" Tadros
This article will explain the different buzz words in the DotNet world like CTS, CLS, VES, Managed Code, Managed Data, Unmanaged Code, Assemblies, Metadata, Manifest, Strong Names, etc....

FIGURE 12.8
The WebDemo program after choosing a page that's well known by Delphi developers

Other useful events are the `OnTitleChange`, used to update the caption with the title of the HTML document, and the `OnStatusTextChange` event, used to update the second part of the status bar. This code basically duplicates the information displayed in the first part of the status bar by the previous two event handlers.

Writing ActiveX Controls

Besides using existing ActiveX controls in Delphi, you can easily develop new ones, using one of two techniques:

- You can use the ActiveX Control Wizard to turn a VCL control into an ActiveX control. You begin from an existing VCL component, which must be a `TWinControl` descendant (and must not have inappropriate properties, in which case it is removed form the combo box of the Wizard), and Delphi wraps an ActiveX around it. During this step, Delphi adds a type library to the control. (Wrapping an ActiveX control around a Delphi component is the opposite of what you did to use an ActiveX control in Delphi.)

- You can create an ActiveForm, place several controls in it, and use the entire form (without borders) as an ActiveX control. This second technique was introduced to build Internet applications, but it is also a very good alternative for constructing an ActiveX control based on multiple Delphi controls or on Delphi components that do not descend from `TWinControl`.

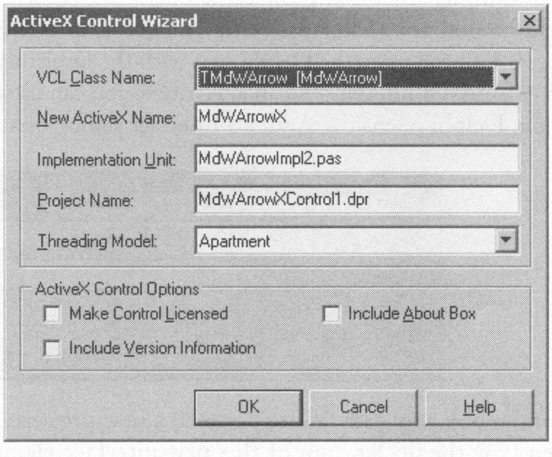
In either case, you can optionally prepare a property page for the control, to use as a sort of property editor for setting the initial value of the control's properties in any development environment—an alternative to the Object Inspector in Delphi. Because most development environments allow only limited editing, it is more important to write a property page than it is to write a component or property editor for a Delphi control.

Building an ActiveX Arrow

As an example of the development of an ActiveX control, I've decided to take the Arrow component from Chapter 9, "Writing Delphi Components," and turn it into an ActiveX control. You cannot use that component directly, because it is a graphical control (a subclass of `TGraphicControl`). However, turning a graphical control into a window-based control is usually a straightforward operation.

In this case, you change the base class name to `TCustomControl` (and change the name of the control's class to `TMdWArrow`, as well, to avoid a name clash), as you can see in the source code files in the `XArrow` folder. After installing this component in Delphi, you are ready to begin developing the new example. To create a new ActiveX library, select File ➢ New ➢ Other, move to the ActiveX page, and choose ActiveX library. Delphi creates the bare skeleton of a DLL, as you saw at the beginning of this chapter. I've saved this library as `XArrow`, in a directory with the same name, as usual.

Now it is time to use the ActiveX Control Wizard, available in the ActiveX page of the Object Repository—Delphi's New dialog box:



VCL Class Name: TMdWArrow [MdWArrow]
New ActiveX Name: MdWArrowX
Implementation Unit: MdWArrowImpl2.pas
Project Name: MdWArrowXControl1.dpr
Threading Model: Apartment

ActiveX Control Options
☐ Make Control Licensed
☐ Include About Box
☐ Include Version Information

In this wizard, you select the VCL class you are interested in, customize the names shown in the edit boxes, and click OK; Delphi then builds the complete source code of an ActiveX control for you.

The use of the three check boxes at the bottom of the ActiveX Control Wizard window may not be obvious. If you make the control licensed, Delphi will include a license key in the code and provide this same GUID in a separate .LIC file. This license file is necessary to use the control in a design environment without the proper *license key* for the control or use it within a web page. The second check box allows you to include version information for the ActiveX control in the OCX file. If the third check box is selected, the ActiveX Control Wizard automatically adds an About box to the control.

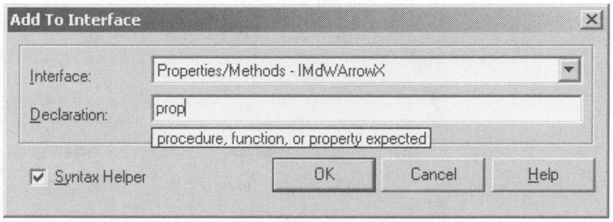
Look at the code the ActiveX Control Wizard generates. This wizard's key element is the generation of a type library and, of course, a corresponding type library import unit with the definition of an interface (`dispinterface`) and other types and constants. In this example, the import file is named `XArrow_TLB.PAS`: I suggest you study it to understand how Delphi defines an ActiveX control. The unit includes a GUID for the control, constants for the definition of values corresponding to the COM enumerated types used by properties of the Delphi control (like `TxMdWArrowDir`), and the declaration of the `IMdWArrowX` interface. The final part of the import unit includes the declaration of the `TMdWArrowX` class. This is a `TOleControl`-derived class you can use to install the control in Delphi, as you saw in the first part of this chapter. You don't need this class to build the ActiveX control; you need it to install the ActiveX control in Delphi.

The rest of the code, and the code you'll customize, is in the main unit, which in the XArrow example is called MdWArrowImpl1. This unit has the declaration of the ActiveX server object, `TMdWArrowX`, which inherits from `TActiveXControl` and implements the specific `IMdWArrowX` interface.

Before you customize this control, let's see how it works. Compile the ActiveX library and then register it using Delphi's Run ➢ Register ActiveX Server menu command. Now you can install the ActiveX control as you've done in the past, except you have to specify a different name for the new class to avoid a name clash. If you use this control, it doesn't look much different from the original VCL control, but the same component can now be installed in other development environments.

Adding New Properties

Once you've created an ActiveX control, adding new properties, events, or methods to it is—surprisingly—simpler than doing the same operation for a VCL component. Delphi, provides specific visual support for adding properties, methods, or events to an ActiveX control, but not for a VCL control. You can open the Delphi unit with the implementation of the ActiveX control, and choose Edit ➢ Add To Interface. As an alternative, you can use the same command from the editor's shortcut menu. Delphi opens the Add To Interface dialog box:


Add To Interface
Interface:
Properties/Methods - IMdWArrowX
Declaration:
prop
procedure, function, or property expected
☑ Syntax Helper
OK
Cancel
Help

In the combo box, you can choose between a new property, method, or event. In the edit box, you can then type the declaration of this new interface element. If the Syntax Helper check box is activated, you'll get hints describing what you should type next and highlighting any errors.

When you define a new ActiveX interface element, keep in mind that you are restricted to COM data types.

In the XArrow example, I've added two properties to the ActiveX control. Because the `Pen` and `Brush` properties of the original Delphi components are not accessible, I've made their color available. These are examples of what you can write in the Add To Interface dialog box's edit box (executing it twice):

```
property FillColor: Integer;
property PenColor: Integer;
```

NOTE *Because a* `TColor` *is a specific Delphi definition, it is not legal to use it.* `TColor` *is an integer subrange that defaults to integer size, so I've used the standard* `Integer` *type directly.*

The declarations you enter in the Add To Interface dialog box are automatically added to the control's type library (TLB) file, to its import library unit, and to its implementation unit. All you have to do to finish the ActiveX control is fill in the `Get` and `Set` methods of the implementation. If you now install this ActiveX control in Delphi once more, the two new properties will appear. The only problem with this property is that Delphi uses a plain integer editor, making it difficult to enter the value of a new color by hand. A program, by contrast, can easily use the `RGB` function to create the proper color value.

Adding a Property Page

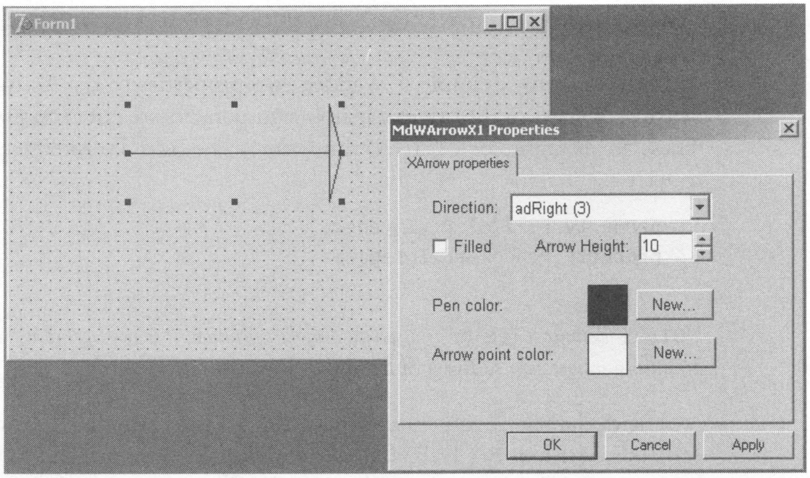
As it stands, other development environments can do very little with your component, because you've prepared no property page—no property editor. A property page is fundamental so that programmers using the control can edit its attributes. However, adding a property page is not as simple as adding a form with a few controls. The property page, will integrate with the host development environment. The property page for your control will show up in a property page dialog of the host environment, which will provide the OK, Cancel, and Apply buttons, and the tabs for showing multiple property pages (some of which may be provided by the host environment).

The nice thing is that support for property pages is built into Delphi, so adding one takes little time. You open an ActiveX project, then open the usual New Items dialog box, move to the ActiveX page, and choose Property Page. What you get is not very different from a form—the `TPropertyPage1` class (created by default) inherits from the `TPropertyPage` class of VCL, which in turn inherits from `TCustomForm`.

TIP *Delphi provides four built-in property pages for colors, fonts, pictures, and strings. The GUIDs of these classes are indicated by the constants* `Class_DColorPropPage`, `Class_DFontPropPage`, `Class_DPicturePropPage`, *and* `Class_DStringPropPage` *in the AxCtrls unit.*

In the property page, you can add controls as in a normal Delphi form, and you can write code to let the controls interact. In the XArrow example, I've added to the property page a combo box with the possible values of the `Direction` property, a check box for the `Filled` property, an edit box with an UpDown control to set the `ArrowHeight` property, and two shapes with corresponding buttons for the colors. You can see this form in the Delphi IDE while working on the ActiveX control in Figure 12.9.

FIGURE 12.9
The XArrow ActiveX control and its property page, hosted by the Delphi environment



Direction: adRight (3)

☐ Filled

Arrow Height: 10

The only code added to the form relates to the two buttons used to change the color of the two shape components, which offer a preview of the ActiveX control's color. The button's `OnClick` event uses a ColorDialog component, as usual:

```
procedure TPropertyPage1.ButtonPenClick(Sender: TObject);
begin
  with ColorDialog1 do
  begin
    Color := ShapePen.Brush.Color;
    if Execute then
    begin
      ShapePen.Brush.Color := Color;
      Modified; // enable Apply button!
    end;
  end;
end;
```

It's important to notice in this code the call to the `Modified` method of the `TPropertyPage` class. This call is required to let the property page dialog box know you've modified one of the values and to enable the Apply button. When a user interacts with one of the other controls on the form, this `Modified` call is made automatically to the `TPropertyPage` class method that handles the internal `cm_Changed` message. As a user, you don't *change* the buttons for these controls, however, you need to add this line yourself.

TIP *Another tip relates to the* `Caption` *of the property page form. It will be used in the host environment's property dialog box as the caption of the tab corresponding to the property page.*

The next step is to associate the property page's controls with the ActiveX control's properties. The property page class automatically has two methods for this functionality: `UpdateOleObject` and

`UpdatePropertyPage`. As their names suggest, these methods copy data from the property page to the ActiveX control and vice versa, as you can see in the example code.

The final step is to connect the property page to the ActiveX control. When the control was created, the Delphi ActiveX Control Wizard automatically added a declaration for the `DefinePropertyPages` method to the implementation unit. In this method, you call the `DefinePropertyPage` method (this time the method name is singular) for each property page you want to add to the control. The parameter of this method is the GUID of the property page, which you can find in the corresponding unit:

```
procedure TMdWArrowX.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

You've finished developing the property page. After recompiling and reregistering the ActiveX library, you can install the ActiveX control in a host development environment (including Delphi) and see how it looks, as I already did in Figure 12.9.

ActiveForms

As I've mentioned, Delphi provides an alternative to the use of the ActiveX Control Wizard to generate an ActiveX control. You can use an ActiveForm, which is an ActiveX control that is based on a form and can host one or more Delphi components. This technique is used in Visual Basic to build new controls, and it makes sense when you want to create a compound component.

In the XClock example, I've placed on an ActiveForm a label (a graphic control that cannot be used as a starting point for an ActiveX control) and a timer, and connected the two with a little code. The form/control becomes a container of other controls, which makes it easy to build compound components (easier than for a VCL compound component).

To build such a control, select the ActiveForm icon in the ActiveX page of the File ➢ New dialog box. Delphi will ask you for some information in the ActiveForm Wizard dialog box, which is similar to the ActiveX Control Wizard dialog box.

ACTIVEFORM INTERNALS

Before you continue with the example, let's look at the code generated by the ActiveForm Wizard. The key difference from a plain Delphi form is in the declaration of the new form class, which inherits from the `TActiveForm` class and implements a specific ActiveForm interface. The code generated for the active form class implements quite a few `Set` and `Get` methods, which change or return the corresponding properties of the Delphi form; this code also implements the events, which again are the events of the form.

The `TForm` events are set to the internal methods when the form is created. For example:

```
procedure TAXForm1.Initialize;
begin
  OnActivate := ActivateEvent;
  ...
end;
```

Each event then maps itself to the external ActiveX event, as in the following method:

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;
```

Because of this mapping, you should not handle the form's events directly. Instead, you can either add code to these default handlers or override the `TForm` methods that end up calling the events. This mapping problem relates only to the events of the form itself, not to the events of the form's components. You can continue to handle the components' events as usual.

NOTE These problems (and possible solutions) are demonstrated by the XForm1 example. I won't discuss it in detail, but leave it for you as a self-study example.

THE XCLOCK ACTIVEX CONTROL

Now that I've covered some foundations, let's return to the development of the XClock example:

1. Place on the form a timer and a label with a large font and centered text, aligned to the client area.
2. Write an event handler for the timer's `OnTimer` event, so that the control updates the output of the label with the current time every second:

   ```
   procedure TXClock.Timer1Timer(Sender: TObject);
   begin
     Label1.Caption := TimeToStr (Time);
   end;
   ```

3. Compile this library, register it, and install it in a package to test it in the Delphi environment.

Notice the effect of the sunken border. This is controlled by the active form's `AxBorderStyle` property, one of the few properties of active forms that is not available for a plain form.

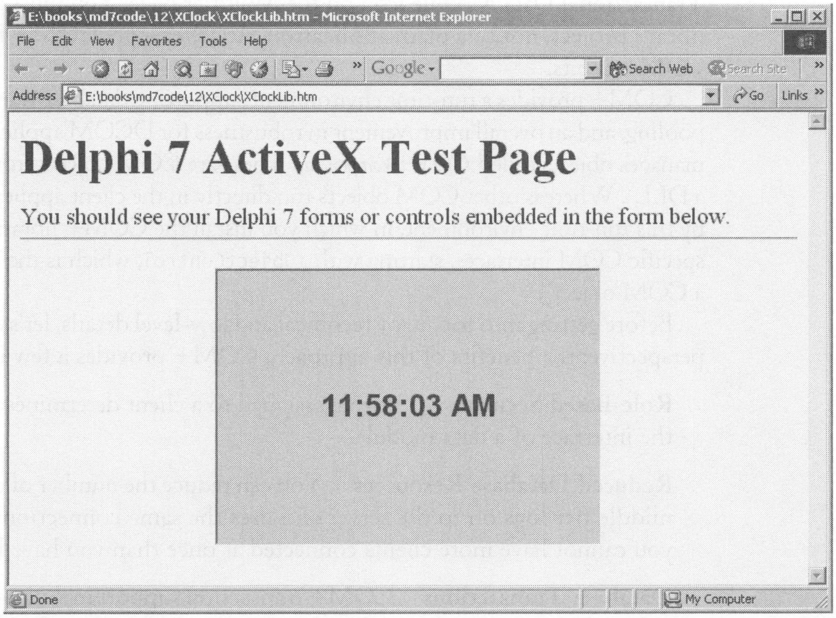
ActiveX in Web Pages

In the previous example, you used Delphi's ActiveForm technology to create a new ActiveX control. An ActiveForm is an ActiveX control based on a form. Borland documentation often implies that ActiveForms should be used in HTML pages, but you can use any ActiveX control on a web page. Basically, each time you create an ActiveX library, Delphi should enable the Project ➢ Web Deployment Options and Project ➢ Web Deploy menu items.

WARNING Due to what I consider a bug, in Delphi 7 these commands are activated only for an ActiveForm. If they are disabled, you can use a trick: Add an ActiveForm to your current ActiveX library, which will enable the menu commands; then immediately remove the ActiveForm, and the menu items will still be available. The trouble is, you'll have to repeat this operation every time you reopen the project—that is, until Borland fixes the bug.

The first command allows you to specify how and where to deliver the proper files. In this dialog box you can set the server directory for deploying the ActiveX component, the URL of this directory, and the server directory for deploying the HTML file (which will have a reference to the ActiveX library using the URL you provide).

You can also specify the use of a compressed CAB file, which can store the OCX file and other auxiliary files, such as packages, making it easier and faster to deliver the application to the user. A compressed file, means a faster download. I've generated the HTML file and CAB file for the XClock project in the same directory. Opening this HTML file in Internet Explorer produces the output shown in Figure 12.10. If all you get is a red *X* marker indicating a failure to download the control, there are various possible explanations for this problem: Internet Explorer doesn't allow the download of controls, it doesn't match the security level for the unsigned control, there is a mismatch in the control version number, and so on.


E:\books\md7code\12\XClock\XClockLib.htm - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Google
Search Web
Search Site
Address E:\books\md7code\12\XClock\XClockLib.htm
Go
Links
Done
My Computer

Delphi 7 ActiveX Test Page

You should see your Delphi 7 forms or controls embedded in the form below.

11:58:03 AM

FIGURE 12.10
The XClock control in the sample HTML page

Notice that in the portion of the HTML file referring to the control, you can use the special `param` tag to customize the control's properties. For example, in the XArrow control's HTML file, I've modified the automatically generated HTML file (in the XArrowCust.htm file) with these three `param` tags:

```
<object classid="clsid:482B2145-4133-11D3-B9F1-00000100A27B"
  codebase="./XArrow.cab" width="350" height="250" align="center"
  hspace="0" vspace="0">
  <param name="ArrowHeight" value="100">
  <param name="Filled" value="-1">
  <param name="FillColor" value="111829">
</object>
```

Although this might seem to be a useful technique, it is important to consider the (limited) role of an ActiveX form placed in a web page. It corresponds to letting a user download and execute a custom Windows application, which raises many concerns about security. An ActiveX control can access the computer's system information, such as the user name, directory structure, and so on. I could continue, but my point is clear.

Introducing COM+

In addition to plain COM servers, Delphi also allows you to create enhanced COM objects, including stateless objects and transaction support. Microsoft first introduced this type of COM object with the MTS (Microsoft Transaction Server) acronym in Windows NT and 98, and later renamed it COM+ in Windows 2000/XP (I'll call it COM+, but I'm referring to both MTS and COM+).

Delphi supports building both standard stateless objects and DataSnap remote data modules based on stateless objects. In both cases, you begin development by using one of the available Delphi wizards, using the New Items dialog box and selecting the Transactional Object icon on the ActiveX page or the Transactional Data Module icon on the Multitier page. You must add these objects to an ActiveX library project, not to a plain application. Another icon, COM+ Event Object, is used to support COM+ events.

COM+ provides a run-time environment supporting database transaction services, security, resource pooling, and an overall improvement in robustness for DCOM applications. The run-time environment manages objects called *COM+ components*. These are COM objects stored in an in-process server (that is, a DLL). Whereas other COM objects run directly in the client application, COM+ objects are handled by this run-time environment, in which you install the COM+ libraries. COM+ objects must support specific COM interfaces, starting with `IObjectControl`, which is the base interface (like `IUnknown` for a COM object).

Before getting into too many technical and low-level details, let's consider COM+ from a different perspective: the benefits of this approach. COM+ provides a few interesting features, including:

Role-Based Security The role assigned to a client determines whether it has the right to access the interface of a data module.

Reduced Database Resources You can reduce the number of database connections, because the middle tier logs on to the server and uses the same connections for multiple clients (although you cannot have more clients connected at once than you have licenses for the server).

Database Transactions COM+ transaction support includes operations on multiple databases, although few SQL servers other than Microsoft's support COM+ transactions.

Creating a COM+ Component

The starting point for creating a COM+ component is the creation of an ActiveX library project. Then, follow these steps:

1. Select a new Transactional Object in the ActiveX page of the New Items dialog box.
2. In the resulting dialog box (see Figure 12.11), enter the name of the new component (*ComPlus1Object* in my ComPlus1 example).

   The New Transactional Object dialog box allows you to enter a name for the class of the COM+ object, the threading model (because COM+ serializes all the requests, Single or Apartment will generally do), and a transactional model:

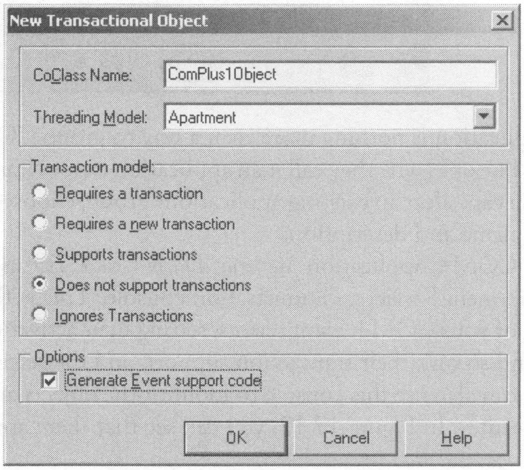
   Requires a Transaction Indicates that each call from the client to the server is considered a transaction (unless the caller supplies an existing transaction context).

Requires a New Transaction Indicates that each call is considered a new transaction.

Supports Transactions Indicates that the client must explicitly provide a transaction context.

Does Not Support Transaction (The default choice, and the one I've used.) Indicates that the remote data module won't be involved in any transaction. This option prevents the object from being activated if the client calling it has a transaction.

Ignores Transactions Indicates that objects do not participate in transactions, but can be used regardless of whether the client has a transaction.



New Transactional Object

CoClass Name: ComPlus1Object

Threading Model: Apartment

Transaction model:

○ Requires a transaction

○ Requires a new transaction

○ Supports transactions

● Does not support transactions

○ Ignores Transactions

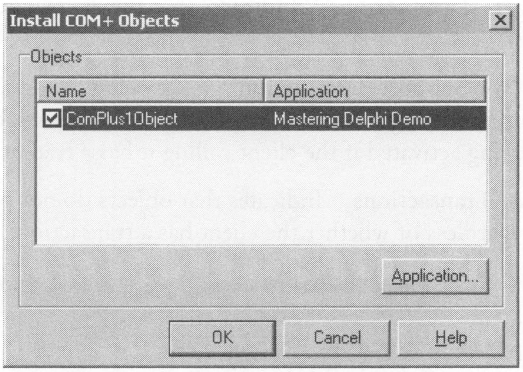
Options

☑ Generate Event support code

FIGURE 12.11
The New Transactional Object dialog box, used to create a COM+ object

3. As you close this dialog, Delphi adds a type library and an implementation unit to the project and opens the type-library editor, where you can define the interface of your new COM object. For this example, add a `Value` integer property, an `Increase` method having as its parameter an amount, and an `AsText` method returning a WideString with the formatted value.
4. As you accept the edits in the type-library editor (by clicking the Refresh button or closing the window), Delphi shows the Implementation File Update Wizard, but only if you set the *Display updates before refreshing* option of the Type Library page of the Environment Options dialog box. This wizard asks for your confirmation before adding four methods to the class, including the get and set methods of the property. You can now write some code for the COM object, which in my example is quite trivial.

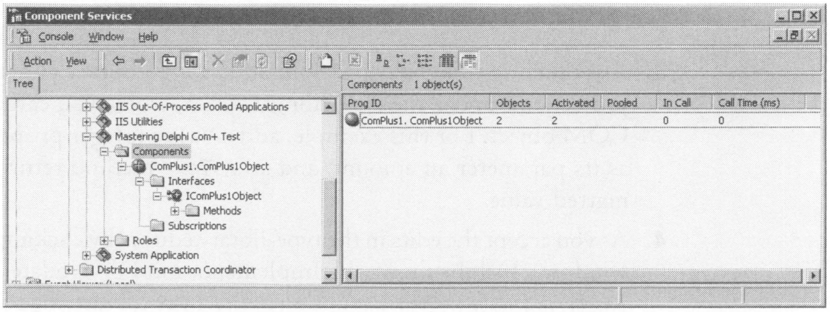
Once you've compiled an ActiveX library, or COM library, which hosts a COM+ component, you can use the Component Services administrative tool (shown in the Microsoft Management Console, or MMC) to install and configure the COM+ component. Even better, you can use the Delphi IDE to install the COM+ component using the Run ➢ Install COM+ Object menu command. In the subsequent dialog box, you can select the component to install (a library can host multiple components) and choose the COM+ application where you want to install the component:


Install COM+ Objects
Objects
Application...
OK
Cancel
Help

Name	Application
☑ ComPlus1Object	Mastering Delphi Demo

A COM+ application is nothing more than a way to group COM+ components; it is not a program or anything like one (why they call it an application is not clear to me). So, in the Install COM+ Object dialog, you can select an existing application/group, choose the Install Into New Application page, and enter a name and description.

I've called the COM+ application *Mastering Delphi Com+ Test,* as you can see in Figure 12.12 in Microsoft's Component Services administration console. This is the front end you can use to fine-tune the behavior of your COM+ components, setting their activation model (just-in-time activation, object pooling, and so on), their transaction support, and the security and concurrency models you want to use. You can also use this console to monitor the objects and method calls (in case they take a long time to execute). In Figure 12.12, you can see that there are currently two active objects.


Component Services
Console
Window
Help
Action
View
Tree
IIS Out-Of-Process Pooled Applications
IIS Utilities
Mastering Delphi Com+ Test
Components
ComPlus1.ComPlus1Object
Interfaces
IComPlus1Object
Methods
Subscriptions
Roles
System Application
Distributed Transaction Coordinator
Components 1 object(s)

Prog ID	Objects	Activated	Pooled	In Call	Call Time (ms)
ComPlus1. ComPlus1Object	2	2		0	0

FIGURE 12.12
The newly installed COM+ component in a custom COM+ application (as shown by Microsoft's Component Services tool)

WARNING *Because you've created one or more objects, the COM library remains loaded in the COM+ environment and some of the objects may be kept in cache, even if no clients are connected to them. For this reason, you generally cannot recompile the COM library after using it, unless you use the MMC to shut it down or set a Transaction Timeout of 0 seconds in MMC.*

I've created a client program for the COM+ object, but it is like any other Delphi COM client. After importing the type library, which is automatically registered while installing the component, I created an interface-type variable referring to it and called its methods as usual.

Transactional Data Modules

The same types of features are available when you create a *transactional data module*—a remote data module within a COM+ component. Once you've created a transactional data module, you can build a Delphi DataSnap application (as you'll see in Chapter 16, "Multitier DataSnap Applications"). You can add one or more dataset components, add one or more providers, and export the provider(s). You can also add custom methods to the data module type library by editing the type library or using the Add To Interface command.

Within a COM+ component or transactional data module, you can also use specific methods that support transactions. These methods are technically provided, at a lower level, in the `IObjectContext` interface returned by the `GetObjectContext` method:

- `SetComplete` tells the COM+ environment the object has finished working and can be deactivated, so that the transaction can be committed.
- `EnableCommit` indicates that the object hasn't finished but the transaction should be committed.
- `DisableCommit` stops the commit operation, even if the method is done, disabling the object deactivation between method calls.
- `SetAbort` says the object has finished and can be activated but the transaction cannot be committed.
- `IsInTransaction` checks whether the object is part of a transaction.

Other methods of the `IContextObject` interface include `CreateInstance`, which creates another COM+ object in the same context and within the current transaction; `IsCallerInRole`, which checks if the object's caller is in a particular "security" role; and `IsSecurityEnabled` (whose name is self-explanatory).

Once you've built a transactional data module within a server library, you can install it as I showed earlier for a plain COM+ object. After the transactional data module has been installed, it will be directly available to other applications and visible in the management console.

An important feature of COM+ is that it becomes much easier to configure DCOM support using this environment. A client computer's COM+ environment can grab information from a server computer's COM+ environment, including registration information for the COM+ object you want to be able to call over a network. The same network configuration is much more complex if done with plain DCOM, without MTS or COM+.

TIP *Even though COM+ configuration is much better than DCOM configuration, you are limited to computers with a recent version of the Windows operating system. Considering that even Microsoft is moving away from DCOM technology, before you build a large system based on it you should evaluate the alternative provided by SOAP (discussed in Chapter 22, "Using XML Technologies").*

COM+ Events

Client applications that use traditional COM objects and Automation servers can call methods of those servers, but this is not an efficient way to check whether the server has updated data for the client. For this reason, a client can define a COM object that implements a *callback* interface, pass this object to the server, and let the server call it. Traditional COM events (which use the `IConnectionPoint` interface) are simplified by Delphi for Automation objects, but are still complex to handle.

COM+ introduces a simplified event model, in which the events are COM+ components and the COM+ environment manages the connections. In traditional COM callbacks, the server object has to keep track of the multiple clients it has to notify, something Delphi doesn't provide us automatically (the default Delphi event code is limited to a single client). To support COM callbacks for multiple clients you need to add the code to hold references to each of the clients. In COM+, the server calls into a single event interface, and the COM+ environment forwards the event to all clients that have expressed interest in it. This way, the client and the server are less coupled, making it possible for a client to receive notification from different servers without any change in its code.

NOTE *Some critics say that Microsoft introduced this model only because it was difficult for Visual Basic developers to handle COM events in the traditional way. Windows 2000 provided a few operating-system features specifically intended for VB developers.*

To create a COM+ event, you should create a COM library (or ActiveX library) and use the COM+ Event Object wizard. The resulting project will contain a type library with the definition of the interface used to fire the events, plus some *fake* implementation code. The server that receives the notification of the events will provide the interface implementation. The fake code is there only to support Delphi's COM registration system.

While building the MdComEvents library, I added to the type library a single method with two parameters, resulting in the following code (in the interface definition file):

```
type
  IMdInform = interface(IDispatch)
    ['{202D2CC8-8E6C-4E96-9C14-1FAAE3920ECC}']
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;
```

The main unit includes the fake COM object (notice that the method is abstract, so it has no implementation) and its class factory, to let the server register itself. At this point, you can compile the library and install it in the COM+ environment, following these steps:

1. In Microsoft's Component Services console, select a COM+ application, move to the Components folder, and use the shortcut menu to add a new component to it.
2. In the COM Component Install Wizard, click the Install New Event Class button and select the library you've just compiled. Your COM+ event definition will be automatically installed.

To test whether it works, you'll have to build an implementation of this event interface and a client invoking it. The implementation can be added to another ActiveX library, hosting a plain COM object. Within Delphi's COM Object Wizard, you can select the interface to implement from the list that appears when you click the List button.

The resulting library, which in my example is called EvtSubscriber, exposes an Automation object: a COM object implementing the `IDispatch` interface (which is mandatory for COM+ events). The object has the following definition and code:

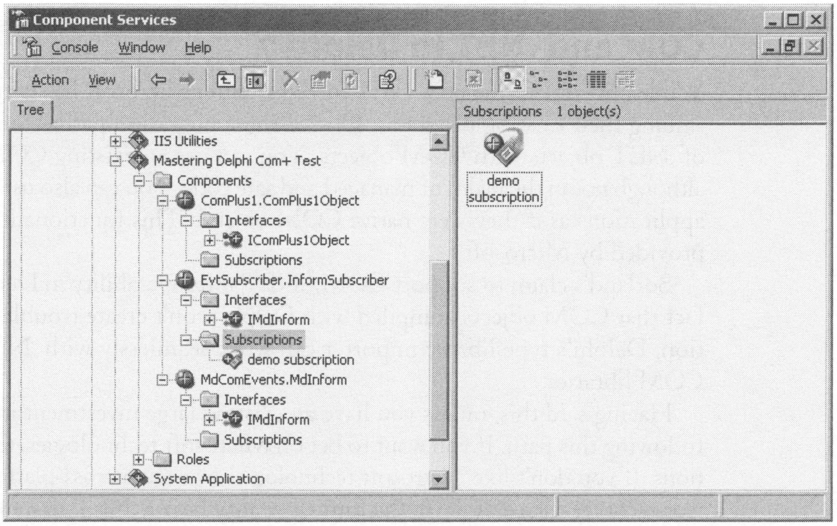
```
type
  TInformSubscriber = class(TAutoObject, IMdInform)
  protected
```

```
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;

procedure TInformSubscriber.Informs(Code: Integer; const Message: WideString);
begin
  ShowMessage ('Message <' + IntToStr (Code) + '>: ' + Message);
end;
```

After compiling this library, you can first install it into the COM+ environment, and then bind it to the event. This second step is accomplished in the Component Services management console by selecting the `Subscriptions` folder under the event object registration, and using the New ➢ Subscription shortcut menu. In the resulting wizard, choose the interface to implement (there is probably only one interface in your COM+ event library); you'll see a list of COM+ components that implement this interface. Selecting one or more of them sets up the subscription binding, which is listed under the `Subscriptions` folder. You can see an example of my configuration while building this example in Figure 12.13.

FIGURE 12.13

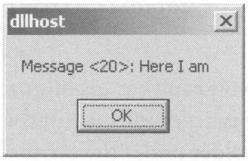
A COM+ event with two subscriptions in the Component Services management console


Component Services
Console
Window
Help
Action
View
Tree
Subscriptions 1 object(s)
IIS Utilities
Mastering Delphi Com+ Test
Components
ComPlus1.ComPlus1Object
Interfaces
IComPlus1Object
Subscriptions
EvtSubscriber.InformSubscriber
Interfaces
IMdInform
Subscriptions
demo subscription
MdComEvents.MdInform
Interfaces
IMdInform
Subscriptions
Roles
System Application
demo subscription

Finally, you can focus on the application that fires the event, which I've called Publisher (because it publishes the information other COM objects are interested in). This is the simplest step of the process, because it is a plain COM client that uses the event server. After importing the COM+ event type library, you can add to the publisher code like this:

```
var
  Inform: IMdInform;
begin
  Inform := CoMdInform.Create;
  Inform.Informs (20, Edit1.Text);
```

My example creates the COM object in the `FormCreate` method to keep the reference around, but the effect is the same. Now the client program thinks it is calling the COM+ event object, but this object (provided by the COM+ environment) calls the method for each of the active subscribers. In this case you'll end up seeing a message box:



Message <20>: Here I am

To make things more interesting, you can subscribe the same server twice to the event interface. The net effect is that without touching your client code, you'll get two message boxes, one for each of the subscribed servers. Obviously this effect becomes interesting when you have multiple different COM components that can handle the event, because you can easily enable and disable them in the management console, changing the COM+ environment without modifying any program code.

COM and .NET in Delphi 7

While putting out the new .NET infrastructure, Microsoft has tried to help companies that are continuing their existing programs. One of these migration paths is represented by the compatibility of .NET objects with COM objects. You can use an existing COM object in a .NET application, although not in the realm of managed and safe code. You can also use .NET assemblies from Windows applications as if they were native COM objects. This functionality takes place thanks to wrappers provided by Microsoft.

Borland's claim to support COM/.NET interoperability in Delphi 7 is mainly a reference to the fact that COM objects compiled with Delphi won't create trouble for the .NET importer. In addition, Delphi's type library importer can work seamlessly with .NET assemblies as with standard COM libraries.

Having said this, unless you have an existing large investment in COM, I'd discourage you from following this path. If you want to bet on Microsoft technologies, the future lies in native .NET solutions. If you don't like Microsoft technologies or want a cross-platform solution, COM will still be a worse choice than .NET (in the future, we may have a .NET framework for other operating systems).

TIP *The steps suggested here should work also in Delphi 6. Delphi 7 adds an automatic import system that at times has troubles with some of the code generated by the compiler of the Delphi for .NET Preview.*

To demonstrate .NET importing features, I've created a .NET library with an interface and a class implementing it. The interface and class resemble those of the FirstCom example discussed at the beginning of this chapter. Here is the code of the library, which must be compiled with the Delphi for .NET preview compiler. You must create an object, or the linker will remove almost everything from your compiled library (*assembly*, in .NET jargon):

```
library NetLibrary;

uses
  NetNumberClass in 'NetNumberClass.pas';
```

```
{$R *.res}

begin
  // create an object to link all of the code
  TNumber.Create;
end.
```

The code is in the NetNumberClass unit, which defines an interface and a class implementing it:

```
type
  INumber = interface
    function GetValue: Integer;
    procedure SetValue (New: Integer);
    procedure Increase;
  end;

  TNumber = class(TObject, INumber)
  private
    fValue: Integer;
  public
    constructor Create;
    function GetValue: Integer;
    procedure SetValue (New: Integer);
    procedure Increase;
  end;
```

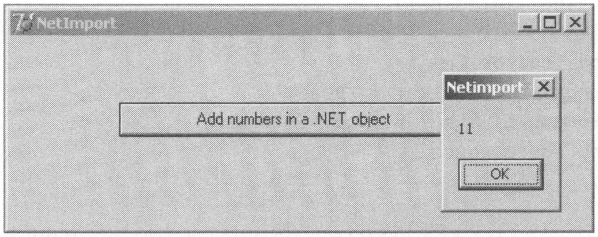
Notice that, unlike a COM server, the interface doesn't require a GUID, following .NET rules (although it can have one using an attribute of the `GuidAttribute` class). The system will generate one for you. After compiling this code (available in the `NetImport` folder of this chapter's code) with Delphi for .NET Preview (not with Delphi 7!), you need to perform two steps: First, run Microsoft's .NET Framework Assembly Registration Utility, *regasm*; second, run Borland's Type Library Importer, *tlibimp*. (In theory, you should be able to skip this step and directly use the Import Type Library dialog box, but with some libraries the use of the *tlibimp* program is required.)

In practice, go to the folder where you've compiled the library and type from the command line the two commands in bold (you should see the rest of the text I've captured here):

```
C:\md7code\NetImport>regasm netlibrary.dll
Microsoft (R) .NET Framework Assembly Registration Utility 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001.  All rights reserved.

Types registered successfully

C:\md7code\NetImport>tlibimp netlibrary.dll
Borland TLIBIMP Version 7.0
Copyright (c) 1997, 2002 Borland Software Corporation
Type library loaded ....
Created E:\books\md7code\12\NetImport\mscorlib_TLB.dcr
Created E:\books\md7code\12\NetImport\mscorlib_TLB.pas
Created E:\books\md7code\12\NetImport\NetLibrary_TLB.dcr
Created E:\books\md7code\12\NetImport\NetLibrary_TLB.pas
```

The effect is to create a unit for the project's type library and a unit for the imported Microsoft .NET Core Library (`mscorlib.dll`). Now you can create a new Delphi 7 application (a standard Win32 program) and use the .NET objects as if they were COM objects. Here is the code from the NetImport example, shown in Figure 12.14:

```
uses
  NetLibrary_TLB;

procedure TForm1.btnAddClick(Sender: TObject);
var
  num: INumber;
begin
  num := CoTNumber.Create as INumber;
  num.Increase;
  ShowMessage (IntToStr (num.GetValue));
end;
```



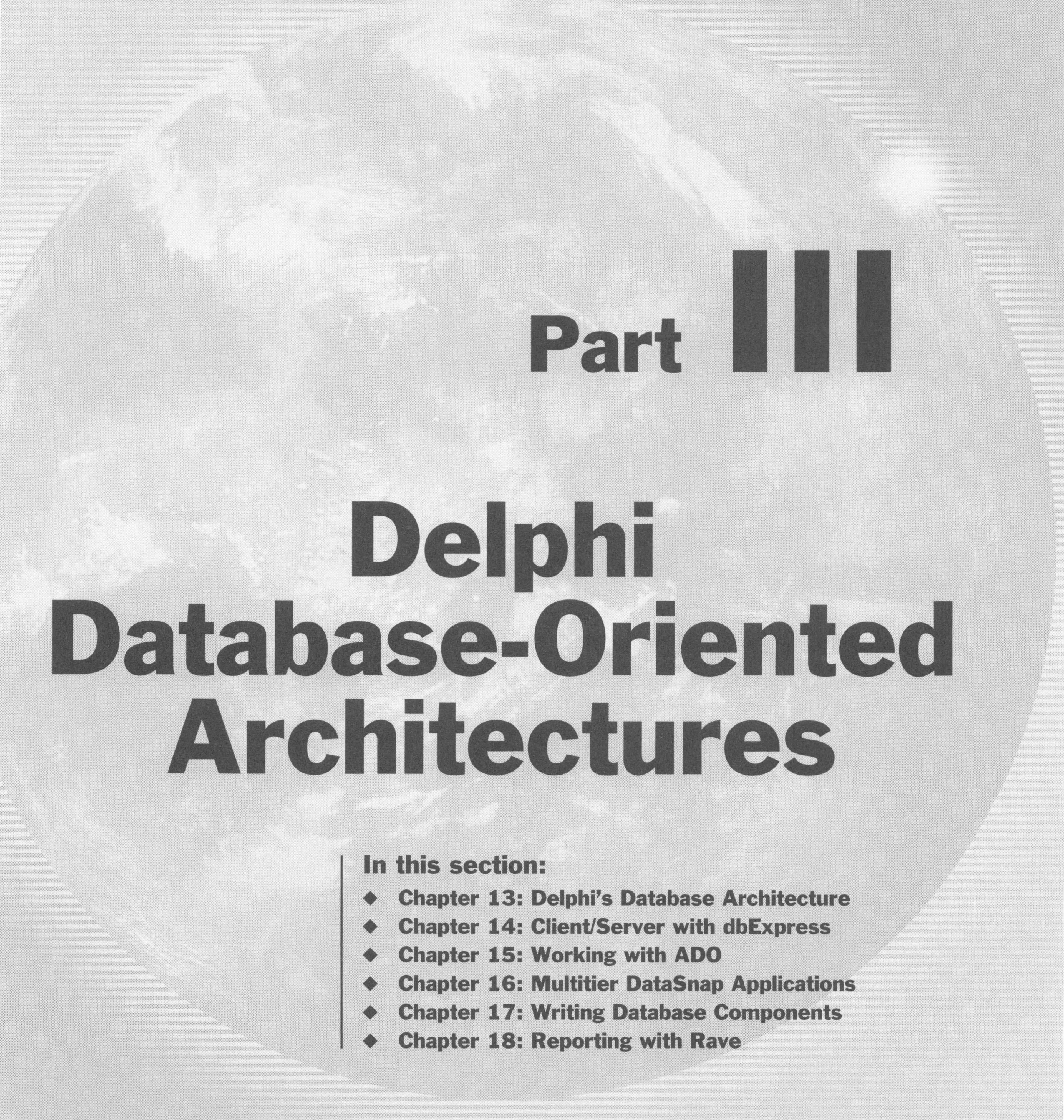
11

FIGURE 12.14
The NetImport program uses a .NET object to sum numbers.

What's Next?

In this chapter, I have discussed applications of Microsoft's COM technology, covering automation, documents, controls, and more. You've seen how Delphi makes the development of Automation servers and clients and ActiveX controls, reasonably simple. Delphi even enables you to wrap components around Automation servers, such as Word and Excel. I've also introduced elements of COM+ and discussed briefly the use of ActiveForms inside a browser. I've stated this is not a good approach to Internet web programming—a topic discussed later in the book.

As I mentioned earlier, if COM has a key role in Windows 2000/XP, future versions of Microsoft's operating systems will downplay its role to push the .NET infrastructure (including SOAP and XML). But you'll have to wait until Chapter 23 for a complete discussion of Delphi SOAP support.



Part III

Delphi Database-Oriented Architectures

In this section:

- Chapter 13: Delphi's Database Architecture
- Chapter 14: Client/Server with dbExpress
- Chapter 15: Working with ADO
- Chapter 16: Multitier DataSnap Applications
- Chapter 17: Writing Database Components
- Chapter 18: Reporting with Rave

Chapter 13

Delphi's Database Architecture

DELPHI'S SUPPORT FOR DATABASE applications is one of the programming environment's key features. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming.

What you *won't* find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports database access.

I'll begin with an explanation of the alternatives Delphi offers in terms of data access, and then I'll provide an overview of the database components available in Delphi. This chapter focuses on the use of the `TClientDataSet` component for accessing local data, deferring all the client/server database access to Chapter 14, "Client/Server with dbExpress." I'll include an overview of the `TDataSet` class, an in-depth analysis of the `TField` components, and the use of data-aware controls. Chapter 14 follows up by providing information about more advanced database programming topics, particularly client/server programming with the use of the dbExpress library (and the InterBase Express components).

Finally, notice that almost everything discussed in this chapter is cross-platform. In particular, the examples can be ported to CLX and to Linux by recompiling and referring to CDS files in the proper folders.

This chapter covers the following topics:

- Delphi's database components
- Database access alternatives
- Using data-aware controls
- The DBGrid control
- Manipulating table fields
- Database applications with standard controls

Accessing a Database: dbExpress, Local Data, and Other Alternatives

The early incarnations of Delphi—immediately adopted as a tool for building database-oriented applications—could access a database only by means of the Borland Database Engine (BDE). Starting with Delphi 3, the portion of VCL related to database access was restructured to open it up to multiple database access solutions, which currently include ADO, native InterBase components, the dbExpress library, and the BDE. Many third-party vendors are now able to offer alternative database access mechanisms to a variety of data formats (some not accessible through Borland components) and still provide a solution integrated with Delphi's VCL.

TIP *In Kylix, the overall picture is slightly different. Borland decided not to port the old BDE technology to Linux and focused instead on the new thin database access layer, dbExpress.*

As a further solution, for simple applications you can use Delphi's ClientDataSet component, which has the ability to save tables to local files—something Borland touts with the name MyBase. Notice that the typical simple Delphi application based on Paradox tables does not port to Kylix, due to the lack of the BDE.

The dbExpress Library

One of the most relevant new features of Delphi in the recent years has been the introduction of the dbExpress database library (DBX), available both for the Linux and the Windows platforms. I say *library* and not *database engine* because, unlike other solutions, dbExpress uses a lightweight approach and requires basically no configuration on end-user machines.

Being light and portable are the two key characteristics of dbExpress; Borland introduced it for those reasons, along with the development of the Kylix project. Compared to other powerhouses, dbExpress is very limited in its capabilities. It can access only SQL servers (no local files); it has no caching capabilities and provides only unidirectional access to the data; and it can natively work only with SQL queries and is unable to generate the corresponding SQL update statements.

At first, you might think these limitations make the library useless. On the contrary: These are *features* that make it interesting. Unidirectional datasets with no direct update are the norm if you need to produce reports, including generating HTML pages showing the content of a database. If you want to build a user interface to edit the data, however, Delphi includes specific components (the ClientDataSet and Provider, in particular) that provide caching and query resolution. These components allow your dbExpress-based application much more control than a separate (monolithic) database engine, which does extra things for you but often does them the way it wants to, not the way you would like.

dbExpress allows you to write an application that, aside from problems with different SQL dialects, can access many different database engines without much code modification. Supported SQL servers include Borland's own InterBase database, the Oracle database server, the MySQL database (which is popular particularly on Linux), the Informix database, IBM's DB2, and Microsoft SQL Server in Delphi 7. A more detailed description of dbExpress, the related VCL components, and many examples of its use will be provided in Chapter 14; the current chapter focuses on database architecture foundations.

TIP *The availability of a dbExpress driver for Microsoft SQL Server in Delphi 7 fills a significant gap. This database is frequently used on the Windows platform, and developers who needed a solution portable among different database servers often had to include support for Microsoft SQL Server. Now there is one less reason to stick with the BDE. Borland has released an update of the SQL Server dbExpress driver shipping with Delphi 7 to fix a couple of bugs.*

The Borland Database Engine

Delphi still ships with the BDE, which allows you to access local database formats (like Paradox and dBase) and SQL servers as well as anything accessible through ODBC drivers. This was the standard database technology in early versions of Delphi, but Borland now considers it *obsolete*. This is particularly true for the use of the BDE to access SQL servers through the SQL Links drivers. Using the BDE to access local tables is still officially supported, simply because Borland doesn't provide a direct migration path for this type of application.

In some cases, a local table can be replaced with the ClientDataSet component (MyBase) specifically for temporary and small lookup tables. However, this approach won't work for larger local tables, because MyBase requires the entire table to be loaded in memory to access even a single record. The suggestion is to move larger tables to a SQL server installed on the client computer. InterBase, with its small footprint, is ideal in this particular situation. This type of migration will also open to you the doors of Linux, where the BDE is not available.

Of course, if you have existing applications that use the BDE, you can continue using them. The BDE page of Delphi's Component Palette still has the Table, Query, StoreProc, and other BDE-specific components. I'd discourage you from developing new programs with this old technology, which is almost discontinued by its producer. Eventually, you should look to third-party engines to replace the BDE when your programs require a similar architecture (or you need compatibility with older database file formats).

NOTE *This is the reason I've removed any coverage of the BDE in the current edition of this book. This chapter used to be based on the Table and Query components; it has been rewritten to describe the architecture of Delphi database applications using the ClientDataSet component.*

InterBase Express

Borland has made available another set of database access components for Delphi: InterBase Express (IBX). These components are specifically tailored to Borland's own InterBase server. Unlike dbExpress, this is not a server-independent database engine, but a set of components for accessing a specific database server. If you plan to use only InterBase as your back-end database, using a specific set of components can give you more control over the server, provide the best performance, and allow you to configure and maintain the server from within a custom client application.

NOTE *The use of InterBase Express highlights the case of database-specific custom datasets, which are available from third-party vendors for many servers. (There are other dataset components for InterBase, just as there are for Oracle, local or shared dBase files, and many others.)*

You can consider using IBX (or other comparable sets of components) if you are sure you won't change your database and you want to achieve the best performance and control at the expense of flexibility and portability. The down side is that the extra performance and control you gain may be limited. You'll also have to learn to use another set of components with a specific behavior, rather than learn to use a generic engine and apply your knowledge to different situations.

MyBase and the ClientDataSet Component

The ClientDataSet is a dataset accessing data kept in memory. The in-memory data can be temporary (created by the program and lost as you exit it), loaded from a local file and then saved back to it, or imported by another dataset using a Provider component.

Borland indicates that you should use the ClientDataSet component mapped to a file with the name MyBase, to indicate that it can be considered a local database solution. I have trouble with the way Borland marketing has promoted this technology, but it has a place, as I'll discuss in the section "MyBase: Stand-alone ClientDataSet."

Accessing data from a provider is a common approach both for client/server architectures (as you'll see in Chapter 14) and for multitier architectures (discussed in Chapter 16, "Multitier DataSnap Applications"). The ClientDataSet component becomes particularly useful if the data-access components you are using provide limited or no caching, which is the case with the dbExpress engine.

dbGo for ADO

ADO (ActiveX Data Objects) is Microsoft's high-level interface for database access. ADO is implemented on Microsoft's data-access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. ADO is an engine with features comparable to those of the BDE: database server independence supporting local and SQL servers alike, a heavyweight engine, and a simplified configuration (because it is not centralized). Installation should (in theory) not be an issue, because the engine is part of recent versions of Windows. However, the limited compatibility among versions of ADO will force you to upgrade your users' computers to the same version you've used for developing the program. The sheer size of the MDAC (Microsoft Data Access Components) installation, which updates large portions of the operating system, makes this operation far from simple.

ADO offers definite advantages if you plan to use Access or SQL Server, because Microsoft's drivers for its own databases are of better quality than the average OLE DB providers. For Access databases, specifically, using Delphi's ADO components is a good solution. If you plan to use other SQL servers, first check the availability of a good-quality driver, or you might have some surprises. ADO is very powerful, but you have to learn to live with it— it stands between your program and the database, providing services but occasionally also issuing different commands than you may expect. On the negative side, do not even think of using ADO if you plan future cross-platform development; this Microsoft-specific technology is not available on Linux or other operating systems.

In short, use ADO if you plan to work only on Windows, want to use Access or other Microsoft databases, or find a good OLE DB provider for each of the database servers you plan to work with (at the moment, this factor excludes InterBase and many other SQL servers).

ADO components (part of a package Borland calls dbGo) are grouped in the ADO page of the Component Palette. The three core components are ADOConnection (for database connections), ADOCommand (for executing SQL commands), and ADODataSet (for executing requests that return a result set). There are also three compatibility components—ADOTable, ADOQuery, and ADOStoredProc—that you can use to port BDE-based applications to ADO. Finally, the RDSConnection component lets you access data in remote multitier applications.

NOTE *Chapter 15, "Working with ADO," covers ADO and related technologies in detail. Notice that Microsoft is replacing ADO with its .NET version, which is based on the same core concepts. So, using ADO might provide you with a good path toward native .NET applications (although Borland plans to move dbExpress to that platform, too).*

Custom Dataset Components

As a further alternative, you can write your own custom dataset components, or choose one of the many offerings available. Developing custom dataset components is one of the most complex issues of Delphi programming; it's covered in depth in Chapter 17, "Writing Database Components." Reading that material, you'll also learn about the internal workings of the `TDataSet` class.

MyBase: Stand-alone ClientDataSet

If you want to write a single-user database application in Delphi, the simplest approach is to use the ClientDataSet component and map it to a local file. This local file mapping is different from the traditional data mapping to a local file. The traditional approach is to read from the file a record at a time, and possibly have a second file that stores indexes. The ClientDataSet maps an entire table (and possibly a master/detail structure) to the file in its entirety: When a program starts, the entire file is loaded in memory, and then everything is saved at once.

WARNING *This explains why you cannot use this approach in a multiuser or multiapplication situation. If two programs or two instances of the same program load the same ClientDataSet file in memory and modify the data, the last table saved will overwrite changes made by other programs.*

This support for persisting the content of a ClientDataSet was created a few years ago as a way to implement the so-called briefcase model. A user could (and still can) download data from its database server to the client, save some of the data, work disconnected (while traveling with a laptop computer, for example), and finally reconnect to commit the changes.

Connecting to an Existing Local Table

To map a ClientDataSet to a local file, you set its `FileName` property. To build a minimal program (called MyBase1 in the example), all you need is a ClientDataSet component hooked to a CDS file (there are a few in the `Data` folder available under `\Program Files\Common Files\Borland Shared`), a DataSource (more on this later), and a DBGrid control. Hook the ClientDataSet to the DataSource via the DataSource's DataSet property and the DataSource to the DBGrid via the grid's DataSource property, as in Listing 13.1. At this point turn on the `Active` property of the ClientDataSet and you'll have a program showing database data even at design time; see Figure 13.1.

LISTING 13.1: THE DFM FILE OF THE MYBASE1 SAMPLE PROGRAM

```
object Form1: TForm1
  ActiveControl = DBGrid1
  Caption = 'MyBase1'
  OnCreate = FormCreate
  object DBGrid1: TDBGrid
    DataSource = DataSource1
  end
  object DataSource1: TDataSource
    DataSet = cds
  end
  object cds: TClientDataSet
```

```
        FileName = 'C:\Program Files\Common Files\Borland
          Shared\Data\customer.cds'
    end
  end
```

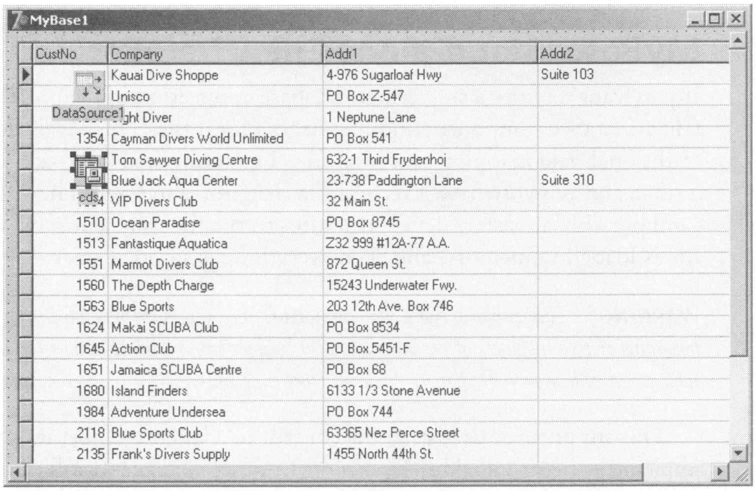


FIGURE 13.1 A sample local table active at design time in the Delphi IDE

CustNo	Company	Addr1	Addr2
	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
	Unisco	PO Box Z-547	
	ght Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
4	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	
1624	Makai SCUBA Club	PO Box 8534	
1645	Action Club	PO Box 5451-F	
1651	Jamaica SCUBA Centre	PO Box 68	
1680	Island Finders	6133 1/3 Stone Avenue	
1984	Adventure Undersea	PO Box 744	
2118	Blue Sports Club	63365 Nez Perce Street	
2135	Frank's Divers Supply	1455 North 44th St.	

As you make changes and close the application, the data will be automatically saved to the file. (You might want to disable the change log, as discussed later, to reduce the size of this data.) The dataset also has a `SaveToFile` method and a `LoadFromFile` method you can use in your code.

I also made another change: I disabled the ClientDataSet at design time to avoid including all of its data in the program's DFM file and in the compiled executable file; I want to keep the data in a separate file. To do this, close the dataset at design time, after testing, and add a line to the form's `OnCreate` event to open it:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  cds.Open;
end;
```

From the Midas DLL to the MidasLib Unit

To run any application using the ClientDataSet component, you need to also deploy the `midas.dll` dynamic library referenced by the `DSIntf.pas` unit. The ClientDataSet component's core code is not directly part of the VCL and is not available in source code format. This is unfortunate, because many developers are accustomed to debugging the VCL source code and using it as the ultimate reference.

WARNING *The* `midas.dll` *library has no version number in its name. So, if a computer has an older version, your program will apparently run on it but may not behave properly.*

The Midas library is a C-language library, but since Delphi 6 it can be bound directly into an executable by including the specific MidasLib unit (a special DCU produced by a C compiler). In this case you won't need to distribute the library in the DLL format.

XML and CDS Formats

The ClientDataSet component supports two different streaming formats: the native format and an XML-based format. The `Borland Shared\Demo` folder mentioned earlier holds versions of a number of tables in each of the two formats. By default, MyBase saves the datasets in XML format. The `SaveToFile` method has a parameter allowing you to specify the format, and the `LoadFromFile` method works automatically for both formats.

Using the XML format has the advantage of making the persistent data also accessible with an editor and with other programs not based on the ClientDataSet component. However, this approach implies converting the data back and forth, because the CDS format is close to the internal memory representation invariably used by the component, regardless of the streaming format. Also, the XML format generates large files, because they are text based. On average, a MyBase XML file is twice the size of the corresponding CDS file.

TIP *While you have a ClientDataSet in memory, you can extract its XML representation by using the* `XMLData` *property without streaming out the data. The next example puts this technique into practice.*

Defining a New Local Table

Besides letting you hook to an existing database table stored in a local file, the ClientDataSet component allows you to create new tables easily. All you have to do is use its `FieldDefs` property to define the structure of the table. After doing this, you can physically create the file for the table with the Create DataSet command on the ClientDataSet component's shortcut menu in the Delphi IDE or by calling its `CreateDataSet` method at run time.

This is an extract from the MyBase2 example's DFM file, which defines a new local database table:

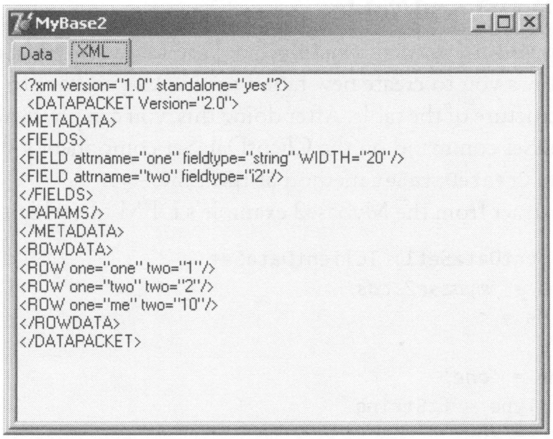
```
object ClientDataSet1: TClientDataSet
  FileName = 'mybase2.cds'
  FieldDefs = <
    item
      Name = 'one'
      DataType = ftString
      Size = 20
    end
    item
      Name = 'two'
      DataType = ftSmallint
    end>
  StoreDefs = True
end
```

Notice the `StoreDefs` property, which is automatically set to True when you edit the collection of field definitions. By default, a dataset in Delphi loads its metadata before opening. Only if a local definition is stored in the DFM file is this local metadata used (saving field definitions in the DFM file is also helpful to cache this metadata in a client/server architecture).

To account for the optional dataset creation, the disabling of the log (described later), and the display of the XML version of the initial data in a Memo control, the program's form class has the following `OnCreate` event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not FileExists (cds.FileName) then
    cds.CreateDataSet;
  cds.Open;
  cds.MergeChangeLog;
  cds.LogChanges := False;
  Memo1.Lines.Text := StringReplace (
    Cds.XMLData, '>', '>' + sLineBreak, [rfReplaceAll]);
end;
```

The last statement includes a call to `StringReplace` to provide a poor man's XML formatting: The code adds a new line at the end of each XML tag by adding a new line after the close angle bracket. You can see the table's XML display with a few records in Figure 13.2. You'll learn a lot more about XML in Delphi in Chapter 22, "Using XML Technologies."



```
<?xml version="1.0" standalone="yes"?>
 <DATAPACKET Version="2.0">
<METADATA>
<FIELDS>
<FIELD attrname="one" fieldtype="string" WIDTH="20"/>
<FIELD attrname="two" fieldtype="i2"/>
</FIELDS>
<PARAMS/>
</METADATA>
<ROWDATA>
<ROW one="one" two="1"/>
<ROW one="two" two="2"/>
<ROW one="me" two="10"/>
</ROWDATA>
</DATAPACKET>
```

FIGURE 13.2
The XML display of a CDS file in the MyBase2 example. The table structure is defined by the program, which creates a file for the dataset on its first execution.

Indexing

Once you have a ClientDataSet in memory, you can perform many operations on it. The simplest are indexing, filtering, and searching for records; more complex operations include grouping, defining aggregate values, and managing the change log. Here I'll cover only the simplest techniques; more complex material appears at the end of the chapter.

Filtering a ClientDataSet is a matter of setting the `IndexFieldNames` property. This is often accomplished when the user clicks the field title in a DBGrid component (firing the `OnTitleClick` event), as in the MyBase2 example:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
```

```
begin
  cds.IndexFieldNames := Column.Field.FieldName;
end;
```

Unlike other local databases, a ClientDataSet can have this type of dynamic index without any database configuration because indexes are computed in memory.

TIP *The component also supports indexes based on a calculated field, specifically an* internally calculated *field, available only for this dataset (as I'll describe later in this chapter). Unlike ordinary calculated fields, which are computed every time the record is used, values of internally calculated fields are calculated once and kept in memory. For this reason, indexes consider them plain fields.*

In addition to assigning a new value to the `IndexFieldNames` property, you can define an index using the `IndexDefs` property. Doing so allows you to define several indexes and keep them in memory, switching even faster from one to the other.

TIP *Defining a separate index is the only way to have a descending index, rather than an ascending index.*

Filtering

As with any other dataset, you can use the `Filter` property to specify the inclusion in the dataset of portions of the data the component is bound to. The filtering operation takes place in memory after loading all the records, so this is a way to present less data to the user, not to limit the memory footprint of a large local dataset.

When you're retrieving large amounts of data from a server (in a client/server architecture) you should try to use a proper query so you don't retrieve a large dataset from a SQL server. Filtering up front in the server should generally be your first choice. With local data, you might consider splitting a large number of records into a set of different files, so you can load only those you need and not all of them.

Local filtering in the ClientDataSet can be useful, particularly because the filter expressions you can use with this component are much more extensive than those you can use with other datasets. In particular, you can use the following:

- The standard comparison and logical operators: for example, `Population > 1000 and Area < 1000`
- Arithmetic operators: for example, `Population / Area < 10`
- String functions: for example, `Substring(Last_Name, 1, 2) = 'Ca'`
- Date and time functions: for example, `Year (Invoice_Date) = 2002`
- Others, including a `Like` function, wildcards, and an `In` operator

These filtering capabilities are fully documented in the VCL Help file. You should look for the page "Limiting what records appear" linked from the description of the `Filter` property of the `TClientDataSet` class, or reach it from the Help Contents page following this path down the tree: Developing Database Applications, Using client datasets, Limiting what records appear.

Locating Records

Filtering allows you to limit the records displayed to the program's user, but many times you want to display all the records and only move to a specific one. The `Locate` method does this. If you've never used `Locate`, at first sight the Help file won't be terribly clear. The idea is that you must provide a list of fields you want to search and a list of values, one for each field. If you want to match only one field, the value is passed directly, as in this case (where the search string is in the `EditName` component):

```
procedure TForm1.btnLocateClick(Sender: TObject);
begin
  if not cds.Locate ('LastName', EditName.Text, []) then
    MessageDlg ('"' + EditName.Text + '" not found', mtError, [mbOk], 0);
end;
```

If you search for multiple fields, you have to pass a variant array with the list of values you want to match. The variant array can be created from a constant array with the `VarArrayOf` function or from scratch using the `VarArrayCreate` call. This is a code snippet:

```
cds.Locate ('LastName;FirstName', VarArrayOf (['Cook', 'Kevin']), [])
```

Finally, you can use the same method to look for a record even if you know only the initial portion of the field you are looking for. All you have to do is to add the loPartialKey flag to the `Options` parameter (the third) of the `Locate` call.

NOTE *Using* `Locate` *makes sense when you're working with a local table, but it doesn't port well to client/server applications. On a SQL server, similar client-side techniques imply moving all the data to the client application first (which is generally a bad idea) and then searching for a specific record. You should locate the data with restricted SQL statements. You can still call* `Locate` *after you retrieve a limited dataset. For example, you can search for a customer by name after you select all the customers of a given town or area, obtaining a result set of a limited size. There's more about this topic in Chapter 14, which is devoted to client/server development.*

UNDO AND *SAVEPOINT*

As a user modifies the data in a ClientDataSet component, the updates are stored in a memory area called `Delta`. The reason for keeping track of user changes instead of holding the resulting table is due to the way the updates are handled in a client/server architecture. In this case, the program doesn't have to send the entire table back to the server, but only a list of the user's changes (by means of specific SQL statements, as you'll see in Chapter 14).

Because the ClientDataSet component keeps track of changes, you can reject those changes, removing entries from the delta. The component has an `UndoLastChange` method to accomplish this. The method's `FollowChange` parameter allows you to *follow* the undo operation—the client dataset will move to the record that has been restored by the undo operation. Here is the code you can use to connect to an Undo button:

```
procedure TForm1.ButtonUndoClick(Sender: TObject);
begin
  cds.UndoLastChange (True);
end;
```

An extension of the undo support is the possibility of saving a sort of bookmark of the change log position (the current status) and to restore it later by undoing all successive changes. You can use the `SavePoint` property either to save the number of changes in the log or to reset the log to a past situation. However, you can only remove records from the change log, not reinsert changes. In other words, the `SavePoint` property refers to a position in a log, so it can only go back to a position where there were fewer records! This log position is a number of changes, so if you save the current position, undo some changes, and then do more edits, you won't be able to get back to the position you bookmarked.

TIP *Delphi 7 has a new standard action mapped to the ClientDataSet's Undo operation. Other new actions include Revert and Apply, which you'll need when the component is connected to a dataset accessing a database.*

ENABLING AND DISABLING LOGGING

Keeping track of changes makes sense if you need to send the updated data back to a server database. In local applications with data stored to a MyBase file, keeping this log around can become useless and consumes memory. For this reason, you can disable logging with the `LogChanges` property. This will also stop the undo operations, though.

You can also call the `MergeChangesLog` method to remove all current editing from the change log, and confirm the edits performed so far. Doing so makes sense if you want to keep the undo log around within a single session and then save the final dataset without the keeping the change log.

NOTE *The MyBase2 example disables the change log as discussed here: You can remove that code and re-enable it to see the difference in the size of the CDS file and in the XML text after editing the data.*

Using Data-Aware Controls

Once you set up the proper data-access components, you can build a user interface to let a user view the data and eventually edit it. Delphi provides many components that resemble the usual controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all these components in the Data Controls page of the Delphi Component Palette.

All these components are connected to a data source using the corresponding property, `DataSource`. Some of them relate to the entire dataset, such as the DBGrid and DBNavigator components, and the others refer to a specific field of the data source, as indicated by the `DataField` property. Once you select the `DataSource` property, the `DataField` property editor will contain a list of available values.

Notice that all the data-aware components are unrelated to the data-access technology, provided the data-access component inherits from `TDataSet`. Thus your investment in the user interface is preserved when you change the data-access technology. However, some of the lookup components and extended use of the DBGrid (displaying a lot of data) make sense only when you're working with local data and should generally be avoided in a client/server situation, as you'll see in Chapter 14.

Data in a Grid

The DBGrid is a grid capable of displaying a whole table at once. It allows scrolling and navigation, and you can edit the grid's contents. It is an extension of the other Delphi grid controls.

You can customize the DBGrid by setting its `Options` property's various flags and modifying its `Columns` collection. The grid allows a user to navigate the data using the scrollbars and perform all the major actions. A user can edit the data directly, insert a new record in a given position by pressing the Insert key, append a new record at the end by going to the last record and pressing the Down arrow key, and delete the current record by pressing Ctrl+Del.

The `Columns` property is a collection from which you can choose the table fields you want to see in the grid and set column and title properties (color, font, width, alignment, caption, and so on) for each field. Some of the more advanced properties, such as `ButtonStyle` and `DropDownRows`, can be used to provide custom editors for a grid's cells or a drop-down list of values (indicated in the column's `PickList` property).

DBNavigator and Dataset Actions

DBNavigator is a collection of buttons used to navigate and perform actions on the database. You can disable some of the DBNavigator control's buttons by removing some of the elements of the `VisibleButtons` set property.

The buttons perform basic actions on the connected dataset, so you can easily replace them with your own toolbar, particularly if you use an ActionList component with the predefined database actions provided by Delphi. In this case, you get all the standard behaviors, but you'll also see the various buttons enabled only when their action is legitimate. The advantages of using the actions is that you can display the buttons in the layout you prefer, intermix them with other buttons of the application, and use multiple client controls, including main and popup menus.

TIP *If you use the standard actions, you can avoid connecting them to a specific DataSource component, and the actions will be applied to the dataset connected to the visual control that currently has the input focus. This way, a single toolbar can be used for multiple datasets displayed by a form, which can be very confusing to the user if not considered carefully.*

Text-Based Data-Aware Controls

There are multiple text-oriented components:

DBText Displays the contents of a field that cannot be modified by the user. It is a data-aware Label graphical control. It can be very useful, but users might confuse this control with the plain labels that indicate the content of each field-based control.

DBEdit Lets the user edit a field (change the current value) using an Edit control. At times, you might want to disable editing and use a DBEdit as if it were a DBText, but highlight the fact that this is data coming from the database.

DBMemo Lets the user see and modify a large text field, eventually stored in a memo or BLOB (binary large object) field. It resembles the Memo component and has full editing capabilities, but all the text is rendered in a single font.

List-Based Data-Aware Controls

To let a user choose a value in a predefined list (which reduces input errors), you can use many different components. DBListBox, DBComboBox, and DBRadioGroup are similar, providing a list of strings in the `Items` property, but they do have some differences:

DBListBox Allows selection of predefined items (*closed selection*), but not text input, and can be used to list many elements. Generally it's best to show only about six or seven items to avoid using up too much space on the screen.

DBComboBox Can be used both for closed selection and for user input. The `csDropDown` style of the DBComboBox allows a user to enter a new value, in addition to selecting one of the available values. The component also uses a smaller area on the form because the drop-down list is usually displayed only on request.

DBRadioGroup Presents radio buttons (which permit only one selection), allows only closed selection, and should be used only for a limited number of alternatives. A nice feature of this component is that the values displayed can be those you want to insert in the database, but you can also choose to provide mapping. The values of the user interface (descriptive strings stored in the `Items` property) will map to corresponding values stored in the database (numeric or character-based codes listed in the `Values` property). For example, you can map numeric codes indicating departments to a few descriptive strings:

```
object DBRadioGroup1: TDBRadioGroup
  Caption = 'Department'
  DataField = 'Department'
  DataSource = DataSource1
  Items.Strings = (
    'Sales'
    'Accounting'
    'Production'
    'Management')
  Values.Strings = (
    '1'
    '2'
    '3'
    '4')
end
```

The DBCheckBox component is slightly different; it is used to show and toggle an option, corresponding to a Boolean field. It is a limited list because it has only two possible values plus the undetermined state for fields with null values. You can determine which are the values to send back to the database by setting the `ValueChecked` and `ValueUnchecked` properties of this component.

THE DBAWARE EXAMPLE

The DbAware example highlights the usage of a DBRadioGroup control with the settings discussed in the previous section and a DBCheckBox control. This example is not much more complex than earlier ones, but it has a form with field-oriented data-aware controls, instead of a grid encompassing them all. You can see the example's form at design time in Figure 13.3.

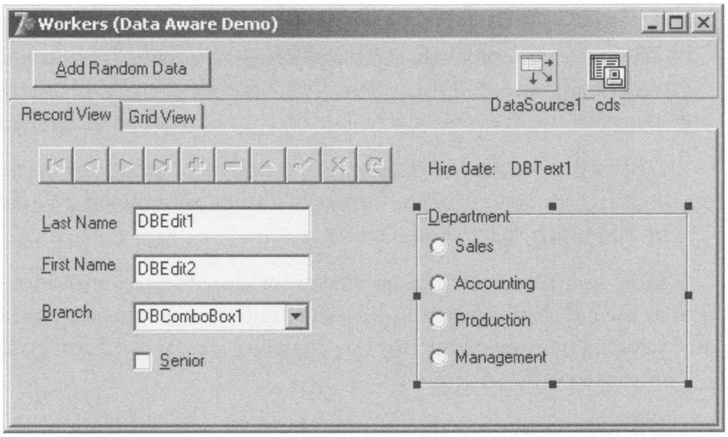

Workers (Data Aware Demo)
Add Random Data
DataSource1
cds
Record View
Grid View
Hire date: DBText1
Last Name
DBEdit1
First Name
DBEdit2
Branch
DBComboBox1
☐ Senior
Department
○ Sales
○ Accounting
○ Production
○ Management

FIGURE 13.3
The data-aware controls of the DbAware example at design time in Delphi

As in the MyBase2 program, the application defines its own table structure, using the `FieldDefs` collection property of the ClientDataSet. Table 13.1 provides a short summary of the fields defined.

TABLE 13.1: THE DATASET FIELDS IN THE DBAWARE EXAMPLE

NAME	DATATYPE	SIZE
LastName	ftString	20
FirstName	ftString	20
Department	FtSmallint	
Branch	ftString	20
Senior	ftBoolean	
HireDate	ftDate	

The program has some code to fill in the table with random values. This code is tedious and not too complex, so I won't discuss the details here, but you can look at the DbAware source code if you are interested.

Using Lookup Controls

If the list of values is extracted from another dataset, then instead of the DBListBox and DBComboBox controls you should use the specific DBLookupListBox or DBLookupComboBox component. These components are used every time you want to select for a field a value that corresponds to a record of another dataset (and not to choose a different record to display!).

For example, if you build a standard form for taking orders, the orders dataset will generally have a field hosting a number indicating the customer who made the order. Working directly with the customer number is not the most natural way; most users will prefer to work with customer names. However, in the database, the customers' names are stored in a different table, to avoid duplicating the customer data for each order by the same customer. To get around such a situation, with local databases or small lookup tables, you can use a DBLookupComboBox control. (This technique doesn't port well to client/server architecture with large lookup tables, as discussed in the next chapter.)

The DBLookupComboBox component can be connected to two data sources at the same time: one source containing the data and a second containing the display data. I built a standard form using the `orders.cds` file from the Delphi sample data folder; the form includes several DBEdit controls.

You should remove the standard DBEdit component connected to the customer number and replace it with a DBLookupComboBox component (and a DBText component, to fully understand what is going on). The lookup component (and the DBText) is connected to the DataSource for the order and to the CustNo field. To let the lookup component show the information extracted from another file (`customer.cds`) you need to add another ClientDataSet component referring to the file, along with a new data source.

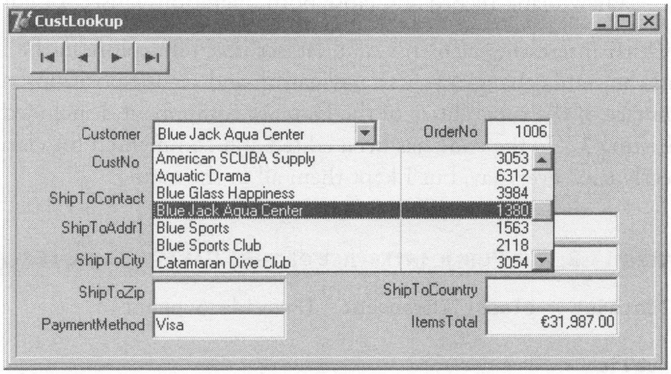
For the program to work, you need to set several properties of the `DBLookupComboBox1` component. Here is a list of the relevant values:

```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSourceOrders
  KeyField = 'CustNo'
  ListField = 'Company;CustNo'
  ListSource = DataSourceCustomer
  DropDownWidth = 300
end
```

The first two properties determine the main connection, as usual. The next four properties determine the field used for the join (`KeyField`), the information to display (`ListField`), and the secondary source (`ListSource`). In addition to entering the name of a single field, you can provide multiple fields, as I did in the example. Only the first field is displayed as combo box text, but if you set a large value for the `DropDownWidth` property, the combo box's drop-down list will include multiple columns of data. You can see this output in Figure 13.4.

FIGURE 13.4

The output of the CustLookup example, with the DBLookupComboBox showing multiple fields in its drop-down list



Customer: Blue Jack Aqua Center

OrderNo: 1006

American SCUBA Supply	3053
Aquatic Drama	6312
Blue Glass Happiness	3984
Blue Jack Aqua Center	1380
Blue Sports	1563
Blue Sports Club	2118
Catamaran Dive Club	3054

PaymentMethod: Visa

ItemsTotal: €31,987.00

TIP *If you set the* `IndexFieldNames` *property of the ClientDataSet containing the orders data to the* `Company` *field, the drop-down list will show the companies in alphabetical order instead of customer-number order. I did this in the example.*

Graphical Data-Aware Controls

Delphi includes a graphical data-aware control: DBImage. It is an extension of an Image component that shows a picture stored in a BLOB field, provided the database uses a graphic format that the Image component supports, such as BMP or JPEG (if you add the JPEG unit to your uses clause).

Once you have a table that includes a BLOB storing an image with a compatible graphic format, hooking it to the component is trivial. If, instead, the graphic format requires a custom transformation in order to be displayed, it might be easier to use a standard non-data-aware Image component and write code so the image is updated each time the current record changes. Before I can discuss this subject, however, you need to know more about the `TDataSet` class and the dataset field classes.

The DataSet Component

Instead of proceeding with the discussion of the capabilities of a specific dataset at this point, I prefer to devote some space to a generic introduction of the features of the `TDataSet` class, which are shared by all inherited data-access classes. The DataSet component is very complex, so I won't list all its capabilities—I will only discuss its core elements.

This component provides access to a series of records that are read from some source of data, kept in internal buffers (for performance reasons), and eventually modified by a user, with the possibility of writing back changes to the persistent storage. This approach is generic enough to be applied to different types of data (even non-database data), but it has a few rules:

- There can be only one active record at a time, so if you need to access data in multiple records, you must move to each of them, read the data, then move again, and so on. You'll find an example of this and related techniques in the section "Navigating a Dataset."
- You can edit only the active record: You cannot modify a set of records at the same time, as you can in a relational database.
- You can modify data in the active buffer only after you explicitly declare you want to do so, by giving the `Edit` command to the dataset. You can also use the `Insert` command to create a new blank record and close both operations (insert or edit) by giving a `Post` command.

Other interesting elements of a dataset that I'll explore in the following sections are its status (and the status change events), navigation and record positions, and the role of field objects. As a summary of the capabilities of the DataSet component, I included the public methods of its class in Listing 13.2 (the code has been edited and commented for clarity). Not all of these methods are directly used everyday, but I kept them all in the listing.

LISTING 13.2: THE PUBLIC INTERFACE OF THE *TDATASET* CLASS (EXCERPTED)

```
TDataSet = class(TComponent, IProviderSupport)
...
public
  // create and destroy, open and close
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure Open;
  procedure Close;
```

```
property BeforeOpen: TDataSetNotifyEvent read FBeforeOpen write FBeforeOpen;
property AfterOpen: TDataSetNotifyEvent read FAfterOpen write FAfterOpen;
property BeforeClose: TDataSetNotifyEvent
  read FBeforeClose write FBeforeClose;
property AfterClose: TDataSetNotifyEvent read FAfterClose write FAfterClose;

// status information
function IsEmpty: Boolean;
property Active: Boolean read GetActive write SetActive default False;
property State: TDataSetState read FState;
function ActiveBuffer: PChar;
property IsUniDirectional: Boolean
  read FIsUniDirectional write FIsUniDirectional default False;
function UpdateStatus: TUpdateStatus; virtual;
property RecordSize: Word read GetRecordSize;
property ObjectView: Boolean read FObjectView write SetObjectView;
property RecordCount: Integer read GetRecordCount;
function IsSequenced: Boolean; virtual;
function IsLinkedTo(DataSource: TDataSource): Boolean;

// datasource
property DataSource: TDataSource read GetDataSource;
procedure DisableControls;
procedure EnableControls;
function ControlsDisabled: Boolean;

// fields, including blobs, details, calculated, and more
function FieldByName(const FieldName: string): TField;
function FindField(const FieldName: string): TField;
procedure GetFieldList(List: TList; const FieldNames: string);
procedure GetFieldNames(List: TStrings); virtual; // virtual since Delphi 7
property FieldCount: Integer read GetFieldCount;
property FieldDefs: TFieldDefs read FFieldDefs write SetFieldDefs;
property FieldDefList: TFieldDefList read FFieldDefList;
property Fields: TFields read FFields;
property FieldList: TFieldList read FFieldList;
property FieldValues[const FieldName: string]: Variant
  read GetFieldValue write SetFieldValue; default;
property AggFields: TFields read FAggFields;
property DataSetField: TDataSetField
  read FDataSetField write SetDataSetField;
property DefaultFields: Boolean read FDefaultFields;
procedure ClearFields;
function GetBlobFieldData(FieldNo: Integer;
  var Buffer: TBlobByteData): Integer; virtual;
function CreateBlobStream(Field: TField;
  Mode: TBlobStreamMode): TStream; virtual;
function GetFieldData(Field: TField;
  Buffer: Pointer): Boolean; overload; virtual;
```

```
procedure GetDetailDataSets(List: TList); virtual;
procedure GetDetailLinkFields(MasterFields, DetailFields: TList); virtual;
function GetFieldData(FieldNo: Integer;
  Buffer: Pointer): Boolean; overload; virtual;
function GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean):
  Boolean; overload; virtual;
property AutoCalcFields: Boolean
  read FAutoCalcFields write FAutoCalcFields default True;
property OnCalcFields: TDataSetNotifyEvent
  read FOnCalcFields write FOnCalcFields;

// position, movement
procedure CheckBrowseMode;
procedure First;
procedure Last;
procedure Next;
procedure Prior;
function MoveBy(Distance: Integer): Integer;
property RecNo: Integer read GetRecNo write SetRecNo;
property Bof: Boolean read FBOF;
property Eof: Boolean read FEOF;
procedure CursorPosChanged;
property BeforeScroll: TDataSetNotifyEvent
  read FBeforeScroll write FBeforeScroll;
property AfterScroll: TDataSetNotifyEvent
  read FAfterScroll write FAfterScroll;

// bookmarks
procedure FreeBookmark(Bookmark: TBookmark); virtual;
function GetBookmark: TBookmark; virtual;
function BookmarkValid(Bookmark: TBookmark): Boolean; virtual;
procedure GotoBookmark(Bookmark: TBookmark);
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; virtual;
property Bookmark: TBookmarkStr read GetBookmarkStr write SetBookmarkStr;

// find, locate
function FindFirst: Boolean;
function FindLast: Boolean;
function FindNext: Boolean;
function FindPrior: Boolean;
property Found: Boolean read GetFound;
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean; virtual;
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant; virtual;

// filtering
property Filter: string read FFilterText write SetFilterText;
property Filtered: Boolean read FFiltered write SetFiltered default False;
property FilterOptions: TFilterOptions
  read FFilterOptions write SetFilterOptions default [];
property OnFilterRecord: TFilterRecordEvent
  read FOnFilterRecord write SetOnFilterRecord;

// refreshing, updating
procedure Refresh;
property BeforeRefresh: TDataSetNotifyEvent
  read FBeforeRefresh write FBeforeRefresh;
property AfterRefresh: TDataSetNotifyEvent
  read FAfterRefresh write FAfterRefresh;
procedure UpdateCursorPos;
procedure UpdateRecord;
function GetCurrentRecord(Buffer: PChar): Boolean; virtual;
procedure Resync(Mode: TResyncMode); virtual;

// editing, inserting, posting, and deleting
property CanModify: Boolean read GetCanModify;
property Modified: Boolean read FModified;
procedure Append;
procedure Edit;
procedure Insert;
procedure Cancel; virtual;
procedure Delete;
procedure Post; virtual;
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
procedure SetFields(const Values: array of const);

// events related to editing, inserting, posting, and deleting
property BeforeInsert: TDataSetNotifyEvent
  read FBeforeInsert write FBeforeInsert;
property AfterInsert: TDataSetNotifyEvent
  read FAfterInsert write FAfterInsert;
property BeforeEdit: TDataSetNotifyEvent read FBeforeEdit write FBeforeEdit;
property AfterEdit: TDataSetNotifyEvent read FAfterEdit write FAfterEdit;
property BeforePost: TDataSetNotifyEvent read FBeforePost write FBeforePost;
property AfterPost: TDataSetNotifyEvent read FAfterPost write FAfterPost;
property BeforeCancel: TDataSetNotifyEvent
  read FBeforeCancel write FBeforeCancel;
property AfterCancel: TDataSetNotifyEvent
  read FAfterCancel write FAfterCancel;
property BeforeDelete: TDataSetNotifyEvent
  read FBeforeDelete write FBeforeDelete;
property AfterDelete: TDataSetNotifyEvent
  read FAfterDelete write FAfterDelete;
property OnDeleteError: TDataSetErrorEvent
  read FOnDeleteError write FOnDeleteError;
  property OnEditError: TDataSetErrorEvent
    read FOnEditError write FOnEditError;
  property OnNewRecord: TDataSetNotifyEvent
    read FOnNewRecord write FOnNewRecord;
  property OnPostError: TDataSetErrorEvent
    read FOnPostError write FOnPostError;

  // support utilities
  function Translate(Src, Dest: PChar;
    ToOem: Boolean): Integer; virtual;
  property Designer: TDataSetDesigner read FDesigner;
  property BlockReadSize: Integer read FBlockReadSize write SetBlockReadSize;
  property SparseArrays: Boolean read FSparseArrays write SetSparseArrays;
end;
```

The Status of a Dataset

When you operate on a dataset in Delphi, you can work in different states. These states are indicated by a specific `State` property, which can assume several different values:

dsBrowse Indicates that the dataset is in normal browse mode; used to look at the data and scan the records.

dsEdit Indicates that the dataset is in edit mode. A dataset enters this state when the program calls the `Edit` method or the DataSource has the `AutoEdit` property set to True, and the user begins editing a data-aware control, such as a DBGrid or DBEdit. When the changed record is posted, the dataset exits the dsEdit state.

dsInsert Indicates that a new record is being added to the dataset. This might happen when calling the `Insert` or `Append` methods, moving to the last line of a DBGrid, or using the corresponding command of the DBNavigator component.

dsInactive Indicates a closed dataset.

dsCalcFields Indicates that a field calculation is taking place (during a call to an `OnCalcFields` event handler).

dsNewValue, dsOldValue, and dsCurValue Indicate that an update of the cache is in progress.

dsFilter Indicates that a dataset is setting a filter (during a call to an `OnFilterRecord` event handler).

In simple examples, the transitions between these states are handled automatically, but it is important to understand them because many events refer to the state transitions. For example, every dataset fires events before and after any state change. When a program requests an `Edit` operation, the component fires the `BeforeEdit` event just before entering edit mode (an operation you can stop by raising an exception). Immediately after entering edit mode, the dataset receives the `AfterEdit` event. After the user has finished editing and requests to store the data by executing the `Post` command, the dataset fires a `BeforePost` event (which can be used to check the input before sending the data to the database); it fires an `AfterPost` event after the operation has been successfully completed.

Another more general state-change tracking technique involves handling the DataSource component's `OnStateChange` event. As an example, you can show the current status with code like this:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  strStatus: string;
begin
  case cds.State of
    dsBrowse: strStatus := 'Browse';
    dsEdit: strStatus := 'Edit';
    dsInsert: strStatus := 'Insert';
  else
    strStatus := 'Other state';
  end;
  StatusBar.Panels[0].Text := strStatus;
end;
```

The code considers only the three most common states of a dataset component, ignoring the inactive state and other special cases.

The Fields of a Dataset

I mentioned earlier that a dataset has only one record that is current, or active. The record is stored in a buffer, and you can operate on it with some generic methods, but to access the data of the record you need to use the dataset's field objects. This explains why field components (technically, instances of a class derived from the `TField` class) play a fundamental role in every Delphi database application. Data-aware controls are directly connected to these field objects, which correspond to database fields.

By default, Delphi automatically creates the `TField` components at run time, each time the program opens a dataset component. This is done after reading the metadata associated with the table or the query the dataset refers to. These field components are stored in the dataset's `Fields` array property. You can access these values by number (accessing the array directly) or by name (using the `FieldByName` method). Each field can be used to read or modify the current record's data by using its `Value` property or type-specific properties such as `AsDate`, `AsString`, `AsInteger`, and so on:

```
var
  strName: string;
begin
  strName := Cds.Fields[0].AsString
  strName := Cds.FieldByName('LastName').AsString
```

`Value` is a variant type property, so using the type-specific access properties is a little more efficient. The dataset component has also a shortcut property for accessing the variant-type value of a field: the default `FieldValues` property. A *default property* means you can omit it from the code by applying the square brackets directly to the dataset:

```
strName := Cds.FieldValues ['LastName'];
strName := Cds ['LastName'];
```

Creating the field components each time a dataset is opened is only a default behavior. As an alternative, you can create the field components at design time, using the Fields Editor (double-click a dataset to see the Fields Editor in action, or activate the dataset's shortcut menu or that of the Object TreeView and choose the Fields Editor command). After creating a field for the LastName column of a table, for example, you can refer to its value by applying one of the `AsXxx` methods to the proper field object:

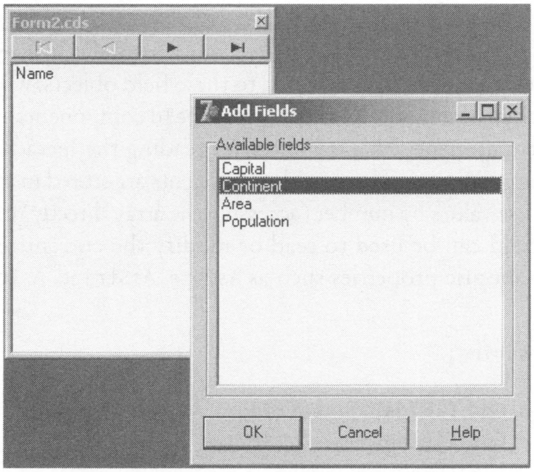
```
strName := CdsLastName.AsString;
```

In addition to being used to access the value of a field, each field object has properties for controlling visualization and editing of its value, including range of values, edit masks, display format, constraints, and many others. These properties, of course, depend on the type of the field—that is, on the specific class of the field object. If you create persistent fields, you can set some properties at design time instead of writing code at run time (perhaps in the dataset's `AfterOpen` event).

NOTE *Although the Fields Editor is similar to the editors of the collections used by Delphi, fields are not part of a collection. They are components created at design time, listed in the published section of the form class and available in the drop-down combo box at the top of the Object Inspector.*

As you open the Fields Editor for a dataset, it appears empty. You have to activate the shortcut menu of this editor or of the Fields pseudonode in the Object TreeView to access its capabilities. The simplest operation you can do is to select the Add command, which allows you to add any other fields in the dataset to the list of fields. Figure 13.5 shows the Add Fields dialog box, which lists all the fields available in a table. These are the database table fields that are not already present in the list of fields in the editor.

FIGURE 13.5
The Fields Editor with the Add Fields dialog box



The Fields Editor's Define command lets you define a new calculated field, lookup field, or field with a modified type. In this dialog box, you can enter a descriptive field name, which might include blank spaces. Delphi generates an internal name—the name of the field component—which you can further customize. Next, select a data type for the field. If it is a calculated field or a lookup field, and not just a copy of a field redefined to use a new data type, select the proper radio button. You'll see how to define a calculated field and a lookup field in the sections, "Adding a Calculated Field" and "Lookup Fields."

NOTE *A* `TField` *component has both a* `Name` *property and a* `FieldName` *property. The* `Name` *property is the usual component name. The* `FieldName` *property is either the name of the column in the database table or the name you define for the calculated field. It can be more descriptive than the* `Name`*, and it allows blank spaces. The* `FieldName` *property of the* `TField` *component is copied to the* `DisplayLabel` *property by default. You can change the field name to any suitable text. It is used, among other things, to search a field in the* `TDataSet` *class's* `FieldByName` *method and when using the array notation.*

All the fields you add or define are included in the Fields Editor and can be used by data-aware controls or displayed in a database grid. If a field of the physical dataset is not in this list, it won't be accessible. When you use the Fields Editor, Delphi adds the declaration of the available fields to the form's class as new components (much as the Menu Designer adds `TMenuItem` components to the form). The components of the `TField` class (more specifically, its subclasses) are fields of the form, and you can refer to these components directly in your program code to change their properties at run time or to get or set their value.

In the Fields Editor, you can also drag the fields to change their order. Proper field ordering is particularly important when you define a grid, which arranges its columns using this order.

TIP *You can also drag the fields from the editor to the form to let the IDE create visual components for you. This is a handy feature that can save you a lot of time when you're creating database-related forms.*

Using Field Objects

Before we look at an example, let's go over the use of the `TField` class. Don't underestimate the importance of this component: Although it is often used behind the scenes, its role in database applications is fundamental. As I already mentioned, even if you do not define specific objects of this kind, you can always access the fields of a table or a query using their `Fields` array property, the `FieldValues` indexed property, or the `FieldByName` method. Both the `Fields` property and the `FieldByName` function return an object of type `TField`, so you sometimes have to use the `as` operator to downcast their result to its type (like `TFloatField` or `TDateField`) before accessing specific properties of these subclasses.

The FieldAcc example has a form with three speed buttons in the toolbar panel, which access various field properties at run time. The first button changes the formatting of the grid's population column. To do this, you have to access the `DisplayFormat` property, which is a specific property of the `TFloatField` class:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
  (cds.FieldByName ('Population') as
    TFloatField).DisplayFormat := '###,###,###';
end;
```

When you set field properties related to data input or output, the changes apply to every record in the table. When you set properties related to the `value` of the field, however, you always refer to the current record only. For example, you can output the population of the current country in a message box by writing the following:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);
begin
  ShowMessage (string (cds ['Name']) +': '+ string (cds ['Population']));
end;
```

When you access the value of a field, you can use a series of `As` properties to handle the current field value using a specific data type (if this data type is available; otherwise, an exception is raised):

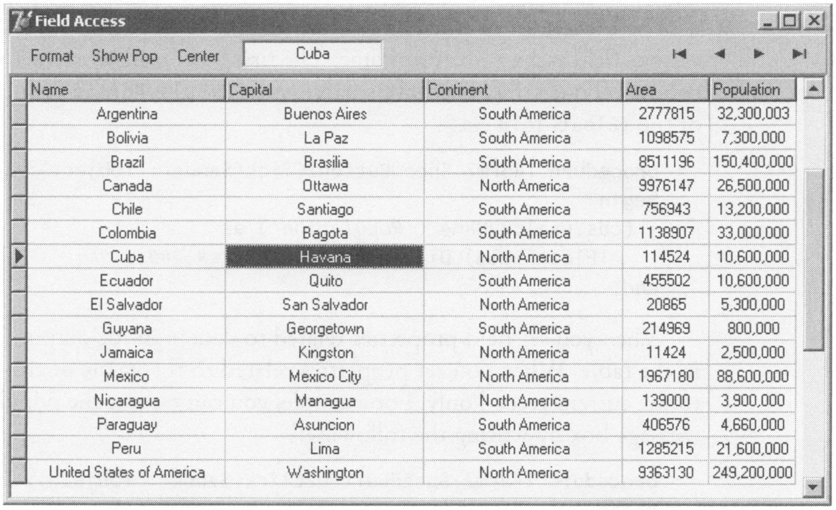
```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt;
AsString: string;
AsVariant: Variant;
```

These properties can be used to read or change the value of the field. Changing the value of a field is possible only if the dataset is in edit mode. As an alternative to the `As` properties, you can access the value of a field by using its `Value` property, which is defined as a variant.

Most of the other properties of the `TField` component, such as `Alignment`, `DisplayLabel`, `DisplayWidth`, and `Visible`, reflect elements of the field's user interface and are used by the various data-aware controls, particularly DBGrid. In the FieldAcc example, clicking the third speed button changes the `Alignment` of every field:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to cds.FieldCount - 1 do
    cds.Fields[I].Alignment := taCenter;
end;
```

This change affects the output of the DBGrid and of the DBEdit control I added to the toolbar, which shows the name of the country. You can see this effect, along with the new display format, in Figure 13.6.



Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32,300,003
Bolivia	La Paz	South America	1098575	7,300,000
Brazil	Brasilia	South America	8511196	150,400,000
Canada	Ottawa	North America	9976147	26,500,000
Chile	Santiago	South America	756943	13,200,000
Colombia	Bagota	South America	1138907	33,000,000
Cuba	Havana	North America	114524	10,600,000
Ecuador	Quito	South America	455502	10,600,000
El Salvador	San Salvador	North America	20865	5,300,000
Guyana	Georgetown	South America	214969	800,000
Jamaica	Kingston	North America	11424	2,500,000
Mexico	Mexico City	North America	1967180	88,600,000
Nicaragua	Managua	North America	139000	3,900,000
Paraguay	Asuncion	South America	406576	4,660,000
Peru	Lima	South America	1285215	21,600,000
United States of America	Washington	North America	9363130	249,200,000

FIGURE 13.6 The output of the FieldAcc example after the Center and Format buttons have been clicked

A Hierarchy of Field Classes

The VCL includes a number of field class types. Delphi automatically uses one of them depending on the data definition in the database, when you open a table at run time or when you use the Fields Editor at design time. Table 13.2 shows the complete list of subclasses of the `TField` class.

TABLE 13.2: THE SUBCLASSES OF TFIELD

SUBCLASS	BASE CLASS	DEFINITION
`TADTField`	`TObjectField`	An ADT (Abstract Data Type) field, corresponding to an object field in an object relational database.
`TAggregateField`	`TField`	Represents a maintained aggregate. It is used in the ClientDataSet component and is discussed in Chapter 14.
`TArrayField`	`TObjectField`	An array of objects in an object relational database.
`TAutoIncField`	`TIntegerField`	A whole positive number connected with a Paradox table's auto-increment field (a special field automatically assigned a different value for each record). Note that Paradox AutoInc fields do not always work perfectly, as discussed in Chapter 14.
`TBCDField`	`TNumericField`	Real numbers with a fixed number of digits after the decimal point.
`TBinaryField`	`TField`	Generally not used directly. This is the base class of the next two classes.
`TBlobField`	`TField`	Binary data with no size limit (BLOB stands for binary large object). The theoretical maximum limit is 2 GB.
`TBooleanField`	`TField`	A Boolean value.
`TBytesField`	`TBinaryField`	Arbitrary data with a large (up to 64 KB characters) but fixed size.
`TCurrencyField`	`TFloatField`	Currency values with the same range as the `Real` data type.
`TDataSetField`	`TObjectField`	An object corresponding to a separate table in an object relational database.
`TDateField`	`TDateTimeField`	A date value.
`TDateTimeField`	`TField`	A date and time value.
`TFloatField`	`TNumericField`	Floating-point numbers (8 byte).
`TFMTBCDField`	`TNumericField`	(New field type in Delphi 6.) A true binary-coded decimal (BCD), as opposed to the existing `TBCDField` type, which converted BCD values to the Currency type. This field type is used automatically only by dbExpress datasets.

Continued on next page

TABLE 13.2: THE SUBCLASSES OF TFIELD *(continued)*

SUBCLASS	BASE CLASS	DEFINITION
`TGraphicField`	`TBlobField`	A graphic of arbitrary length.
`TGuidField`	`TStringField`	A field representing a COM Globally Unique Identifier (part of the ADO support).
`TIDispatchField`	`TInterfaceField`	A field representing pointers to `IDispatch` COM interfaces (part of the ADO support).
`TIntegerField`	`TNumericField`	Whole numbers in the range of long integers (32 bits).
`TInterfacedField`	`TField`	Generally not used directly. This is the base class of fields that contain pointers to interfaces (`IUnknown`) as data.
`TLargeIntField`	`TIntegerField`	Very large integers (64 bit).
`TMemoField`	`TBlobField`	Text of arbitrary length.
`TNumericField`	`TField`	Generally not used directly. This is the base class of all the numeric field classes.
`TObjectField`	`TField`	Generally not used directly. This is the base class for fields providing support for object relational databases.
`TReferenceField`	`TObjectField`	A pointer to an object in an object relational database.
`TSmallIntField`	`TIntegerField`	Whole numbers in the range of integers (16 bits).
`TSQLTimeStampField`	`TField`	(New field type in Delphi 6.) Supports the date/time representation used in dbExpress drivers.
`TStringField`	`TField`	Text data of a fixed length (up to 8192 bytes).
`TTimeField`	`TDateTimeField`	A time value.
`TVarBytesField`	`TBytesField`	Arbitrary data; up to 64 KB of characters. Very similar to the `TBytesField` base class.
`TVariantField`	`TField`	A field representing a variant data type (part of the ADO support).
`TWideStringField`	`TStringField`	A field representing a Unicode (16 bits per character) string.
`TWordField`	`TIntegerField`	Whole positive numbers in the range of words or unsigned integers (16 bits).

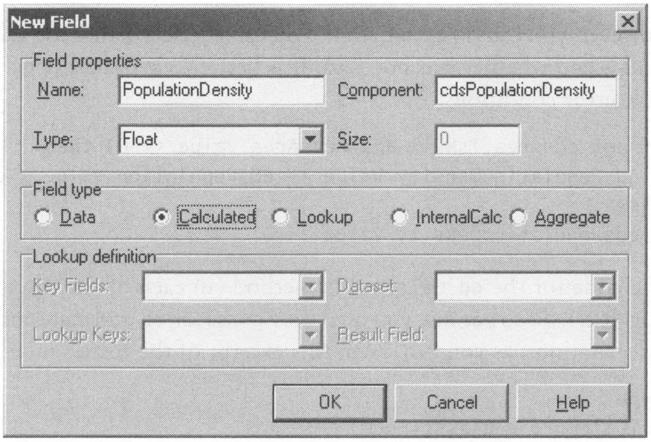
The availability of any particular field type, and the correspondence with the data definition, depends on the database in use. This is particularly true for the field types that provide support for object relational databases.

Adding a Calculated Field

Now that you've been introduced to `TField` objects and have seen an example of their run time use, I will build an example based on the declaration of field objects at design time using the Fields Editor and then add a calculated field. In the `country.cds` sample dataset, both the population and the area of each country are available; you can use this data to compute the population density.

To build the new example, named Calc, follow these steps:

1. Add a ClientDataSet component to a form.
2. Open the Fields Editor. In this editor, right-click, choose the Add Field command, and select some of the fields. (I included them all.)
3. Select the New Field command and enter a proper name and data type (Float, for a `TFloatField`) for the new calculated field, as you can see in Figure 13.7.


New Field

Field properties

Name: PopulationDensity

Component: cdsPopulationDensity

Type: Float

Size: 0

Field type: ○ Data ● Calculated ○ Lookup ○ InternalCalc ○ Aggregate

FIGURE 13.7 The definition of a calculated field in the Calc example

WARNING *Obviously, because you create some field components at design time using the Fields Editor, the fields you skip won't get a corresponding object. As I already mentioned, the fields you skip will not be available even at run time, with* `Fields` *or* `FieldByName`. *When a program opens a table at run time, if there are no design-time field components, Delphi creates field objects corresponding to the table definition. If there are some design-time fields, however, Delphi uses those fields without adding any extra field objects.*

Of course, you also need to provide a way to calculate the new field. This is accomplished in the `OnCalcFields` event of the ClientDataSet component, which has the following code (at least in a first version):

```
procedure TForm2.cdsCalcFields(DataSet: TDataSet);
begin
  cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value;
end;
```

NOTE *In general, calculated fields are computed for each record and recalculated each time the record is loaded in an internal buffer, invoking the* `OnCalcFields` *event over and over. For this reason, a handler of this event should be extremely fast to execute and cannot alter the status of the dataset by accessing different records. A more time-efficient (but less memory-efficient) version of a calculated field is provided by the ClientDataSet component with internally calculated fields: These fields are evaluated only once—when they are loaded—and the result is stored in memory for future requests.*

Everything fine? Not at all! If you enter a new record and do not set the value of the population and area, or if you accidentally set the area to zero, the division will raise an exception, making it problematic to continue using the program. As an alternative, you could have handled every exception of the division expression and set the resulting value to zero:

```
try
  cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value;
except
  on Exception do
    cdsPopulationDensity.Value := 0;
end;
```

However, you can do even better: You can check whether the value of the area is defined—if it is not null—and whether it is not zero. It is better to avoid using exceptions when you can anticipate possible error conditions:

```
if not cdsArea.IsNull and (cdsArea.Value <> 0) then
  cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value
else
  cdsPopulationDensity.Value := 0;
```

The code for the `cdsCalcFields` method (in each of the three versions) accesses some fields directly. It can do so because you used the Fields Editor, and it automatically created the corresponding field declarations, as you can see in this excerpt of the form's interface declaration:

```
type
  TCalcForm = class(TForm)
    cds: TClientDataSet;
    cdsPopulationDensity: TFloatField;
    cdsArea: TFloatField;
    cdsPopulation: TFloatField;
    cdsName: TStringField;
    cdsCapital: TStringField;
    cdsContinent: TStringField;
    procedure cdsCalcFields(DataSet: TDataset);
    ...
```

Each time you add or remove fields in the Fields Editor, you can see the effect of your action immediately in the grid present in the form (unless the grid has its own column objects defined, in which case you often don't see any change). Of course, you won't see the values of a calculated field at design time; they are available only at run time, because they result from the execution of compiled Delphi language code.

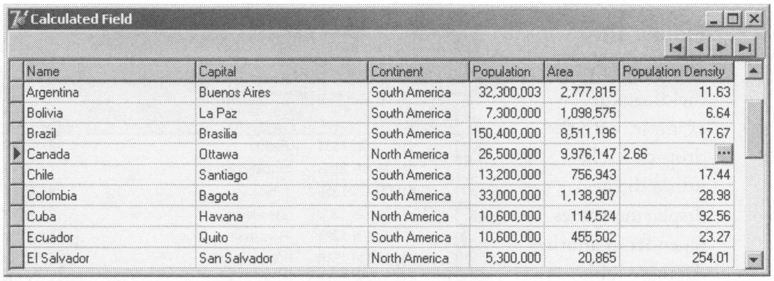
Because you have defined components for the fields, you can use them to customize some of the grid's visual elements. For example, to set a display format that adds a comma to separate thousands, you can use the Object Inspector to change the `DisplayFormat` property of some field components to ###,###,###. This change has an immediate effect on the grid at design time.

NOTE *The display format I just mentioned (and used in the previous example) uses the Windows International Settings to format the output. When Delphi translates the numeric value of this field to text, the comma in the format string is replaced by the proper* `ThousandSeparator` *character. For this reason, the output of the program will automatically adapt itself to different international settings. On computers that have the Italian configuration, for example, the comma is replaced by a period.*

After working on the table components and the fields, I customized the DBGrid using its `Columns` property editor. I set the Population Density column to read-only and set its `ButtonStyle` property to cbsEllipsis to provide a custom editor. When you set this value, a small button with an ellipsis is displayed when the user tries to edit the grid cell. Clicking the button invokes the DBGrid's `OnEditButtonClick` event:

```
procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  MessageDlg (Format (
    'The population density (%.2n)'#13 +
    'is the Population (%.0n)'#13 +
    'divided by the Area (%.0n).'#13#13 +
    'Edit these two fields to change it.',
    [cdsPopulationDensity.AsFloat,
    cdsPopulation.AsFloat,
    cdsArea.AsFloat]),
    mtInformation, [mbOK], 0);
end;
```

I haven't provided a real editor but rather a message describing the situation, as you can see in Figure 13.8, which shows the values of the calculated fields. To create an editor, you might build a secondary form to handle special data entries.



Calculated Field

Name	Capital	Continent	Population	Area	Population Density
Argentina	Buenos Aires	South America	32,300,003	2,777,815	11.63
Bolivia	La Paz	South America	7,300,000	1,098,575	6.64
Brazil	Brasilia	South America	150,400,000	8,511,196	17.67
Canada	Ottawa	North America	26,500,000	9,976,147	2.66
Chile	Santiago	South America	13,200,000	756,943	17.44
Colombia	Bagota	South America	33,000,000	1,138,907	28.98
Cuba	Havana	North America	10,600,000	114,524	92.56
Ecuador	Quito	South America	10,600,000	455,502	23.27
El Salvador	San Salvador	North America	5,300,000	20,865	254.01

FIGURE 13.8
The output of the Calc example. Notice the Population Density calculated column and the ellipsis button displayed when you edit it.

Lookup Fields

As an alternative to placing a DBLookupComboBox component in a form (discussed earlier in this chapter in the section "Using Lookup Controls"), you can also define a lookup field, which can be displayed with a drop-down lookup list inside a DBGrid component. You've seen that to add a fixed selection to a DBGrid, you can edit the `PickList` subproperty of the `Columns` property. To customize the grid with a live lookup, however, you have to define a lookup field using the Fields Editor.

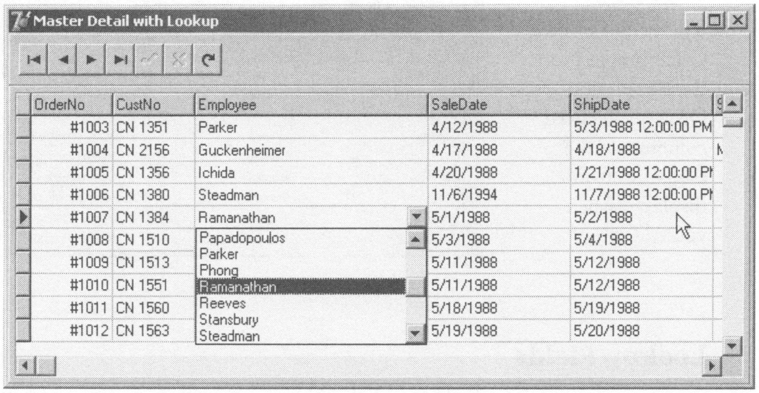
As an example, I built the FieldLookup program, which has a grid that displays orders; it includes a lookup field to display the name of the employee who took the order, instead of the employee's code number. To accomplish this functionality, I added to the data module a ClientDataSet component referring to the `employee.cds` dataset. Then I opened the Fields Editor for the orders dataset and added all the fields. I selected the `EmpNo` field and set its `Visible` property to False to remove it from the grid (you cannot remove it altogether, because it is used to build the cross-reference with the corresponding field of the employee dataset).

Now it is time to define the lookup field. If you followed the preceding steps, you can use the Fields Editor of the orders dataset and select the New Field command to open the New Field dialog box. The values you specify here will affect the properties of a new `TField` added to the table, as demonstrated by the DFM description of the field:

```
object cds2Employee: TStringField
  FieldKind = fkLookup
  FieldName = 'Employee'
  LookupDataSet = cds2
  LookupKeyFields = 'EmpNo'
  LookupResultField = 'LastName'
  KeyFields = 'EmpNo'
  Size = 30
  Lookup = True
end
```

This is all that is needed to make the drop-down list work (see Figure 13.9) and to also view the value of the cross-reference field at design time. Notice that you don't need to customize the `Columns` property of the grid because the drop-down button and the value of seven rows are used by default. However, this doesn't mean you cannot use this property to further customize these and other visual elements of the grid.

FIGURE 13.9
The output of the FieldLookup example, with the drop-down list inside the grid displaying values taken from another database table



OrderNo	CustNo	Employee	SaleDate	ShipDate
#1003	CN 1351	Parker	4/12/1988	5/3/1988 12:00:00 PM
#1004	CN 2156	Guckenheimer	4/17/1988	4/18/1988
#1005	CN 1356	Ichida	4/20/1988	1/21/1988 12:00:00 P
#1006	CN 1380	Steadman	11/6/1994	11/7/1988 12:00:00 P
#1007	CN 1384	Ramanathan	5/1/1988	5/2/1988
#1008	CN 1510		5/3/1988	5/4/1988
#1009	CN 1513		5/11/1988	5/12/1988
#1010	CN 1551		5/11/1988	5/12/1988
#1011	CN 1560		5/18/1988	5/19/1988
#1012	CN 1563		5/19/1988	5/20/1988

Papadopoulos
Parker
Phong
Ramanathan
Reeves
Stansbury
Steadman

This program has another specific feature. The two ClientDataSet components and the two DataSource components have not been placed on a form but rather on a special container for nonvisual components called a *data module* (see the sidebar "A Data Module for Data-Access Components"). You can obtain a data module from Delphi's File ➢ New menu. After adding components to it, you can link them from controls on other forms with the File ➢ Use Unit command.

A DATA MODULE FOR DATA-ACCESS COMPONENTS

To build a Delphi database application, you can place data-access components and the data-aware controls in a form. This approach is handy for a simple program, but having the user interface and the data access and data model in a single (often large) unit is far from a good idea. For this reason, Delphi implements the idea of a *data module*: a container of non-visual components.

At design time, a data module is similar to a form, but at run time it exists only in memory. The `TDataModule` class derives directly from `TComponent`, so it is unrelated to the Windows concept of a window (and is fully portable among different operating systems). Unlike a form, a data module has just a few properties and events. So, it's useful to think of data modules as components and method containers.

Like a form or a frame, a data module has a designer. Delphi creates a specific unit for the definition of the data module's class and a form definition file that lists its components and their properties.

There are several reasons to use data modules. The simplest is that they let you share data-access components among multiple forms, as I'll demonstrate at the beginning of Chapter 14. This technique works in conjunction with visual form linking—the ability to access components of another form or data module at design time (with the File ➢ Use Unit command). The second reason is that data modules separate the data from the user interface, improving the structure of an application. Data modules in Delphi even exist in versions specific for multitier applications (remote data modules) and server-side HTTP applications (web data modules).

Handling Null Values with Field Events

In addition to a few interesting properties, field objects have a few key events. The `OnValidate` event can be used to provide extended validation of a field's value and should be used whenever you need a complex rule that the ranges and constraints provided by the field cannot express. This event is triggered before the data is written to the record buffer, whereas the `OnChange` event is fired soon after the data has been written.

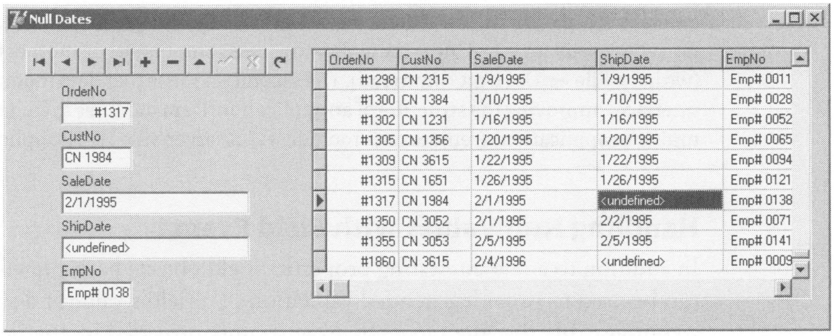
Two other events—`OnGetText` and `OnSetText`—can be used to customize a field's output. These two events are extremely powerful: They allow you to use data-aware controls even when the representation of a field you want to display is different from the one Delphi will provide by default.

Handling null values provides an example of the use of these events. On SQL servers, storing an empty value for a field is a separate operation from storing a null value for a field. The latter tends to be more correct, but Delphi by default uses empty values and displays the same output for an empty or a null field. Although this behavior can be useful in general for strings and numbers, it becomes extremely important for dates, where it is hard to set a reasonable default value and where if the deletes the contents of the field, you might have invalid input.

The NullDates program displays specific text for dates that have a null value and clears the field (setting it to the null value) when the user uses an empty string in input. Here is the relevant code of the field's two event handlers:

```
procedure TForm1.cdsShipDateGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if Sender.IsNull then
    Text := '<undefined>'
    else
      Text := Sender.AsString;
  end;

  procedure TForm1.cdsShipDateSetText(Sender: TField; const Text: String);
  begin
    if Text = '' then
      Sender.Clear
    else
      Sender.AsString := Text;
  end;
```

Figure 13.10 shows an example of the program's output, with undefined (or null) values for some shipping dates.



OrderNo
#1317
CustNo
CN 1984
SaleDate
2/1/1995
ShipDate
<undefined>
EmpNo
Emp# 0138

OrderNo	CustNo	SaleDate	ShipDate	EmpNo
#1298	CN 2315	1/9/1995	1/9/1995	Emp# 0011
#1300	CN 1384	1/10/1995	1/10/1995	Emp# 0028
#1302	CN 1231	1/16/1995	1/16/1995	Emp# 0052
#1305	CN 1356	1/20/1995	1/20/1995	Emp# 0065
#1309	CN 3615	1/22/1995	1/22/1995	Emp# 0094
#1315	CN 1651	1/26/1995	1/26/1995	Emp# 0121
#1317	CN 1984	2/1/1995	<undefined>	Emp# 0138
#1350	CN 3052	2/1/1995	2/2/1995	Emp# 0071
#1355	CN 3053	2/5/1995	2/5/1995	Emp# 0141
#1860	CN 3615	2/4/1996	<undefined>	Emp# 0009

FIGURE 13.10
By handling the *OnGetText* and *OnSetText* events of a date field, the NullDates example displays specific output for null values.

WARNING *The handling of null values in Delphi 6 and 7 can be affected by changes in the way null variants work. As discussed in Chapter 3, "The Run Time Library," in the section "The Variants and VarUtils Units," comparing a field with a null value with another field will have a different effect in the latest versions of Delphi than in the past. As discussed in that section, in Delphi 7 you can use global variables to fine-tune the effect of comparisons involving variants.*

Navigating a Dataset

You've seen that a dataset has only one active record; this active record changes frequently in response to user actions or because of internal commands given to the dataset. To move around the dataset and change the active record, you can use methods of the `TDataSet` class as you saw in Listing 13.2, (particularly in the section commented `position, movement`). You can move to the next or previous record, jump back and forth by a given number of records (with `MoveBy`), or go directly to the first or last record of the dataset. These dataset operations are available in the DBNavigator component and the standard dataset actions, and they are not difficult to understand.

What is not obvious, though, is how a dataset handles extreme positions. If you open any dataset with a navigator attached, you can see that as you move record by record, the Next button remains enabled even when you reach the last record. Only when you try to move forward after the last record does the button become disabled (and the current record doesn't change). This happens because the `Eof` (end of file) test succeeds only when the cursor has been moved to a special position after the last record. If you jump with the Last button instead, you'll immediately be at the end. You'll encounter the same behavior for the first record (and the `Bof` test). As you'll soon see, this approach is handy because you can scan a dataset testing for `Eof` to be True, and know at that point you've already processed the last record of the dataset.

In addition to moving record by record or by a given number of records, programs might need to jump to specific records or positions. Some datasets support the `RecordCount` property and allow movement to a record at a given position in the dataset using the `RecNo` property. You can use these properties only for datasets that support positions natively, which basically excludes all client/server architectures, unless you grab all the records in a local cache (something you'll generally want to avoid) and then navigate on the cache. As you'll see in Chapter 14, when you open a query on a SQL server, you fetch only the records you are using, so Delphi doesn't know the record count (at least, not in advance).

You can use two alternatives to refer to a record in a dataset, regardless of its type:

- You can save a reference to the current record and then jump back to it after moving around. You do so by using bookmarks, either in the `TBookmark` or the more modern `TBookmarkStr` form. This approach is discussed in the section "Using Bookmarks."
- You can locate a dataset record that matches given criteria, using the `Locate` method. This approach, which is presented in the next section, works even after you close and reopen the dataset, because you're working at a logical (not physical) level.

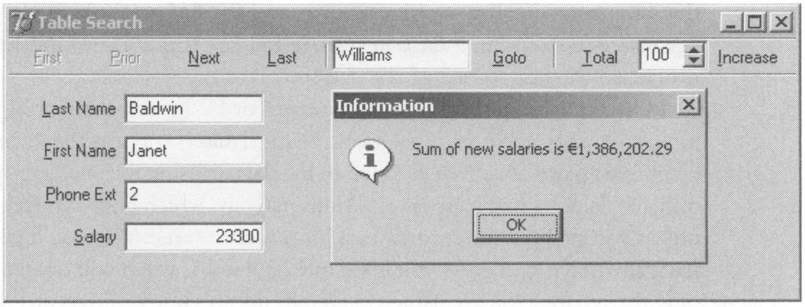
The Total of a Table Column

So far in our examples, the user can view the current contents of a database table and manually edit the data or insert new records. Now you will see how you can change data in the table through the program code. The employee dataset you have already used has a Salary field, so a manager of the company can browse through the table and change the salary of a single employee. But what is the company's total salary expense? And what if the manager wants to give everyone a 10 percent salary increase (or decrease)?

The program, which also demonstrates the use of an action list for the standard dataset actions, has buttons to calculate the sum of the current salaries and change them. The total action lets you calculate the sum of the salaries of all the employees. Basically, you need to scan the table, reading the value of the `cdsSalary` field for each record:

```
var
  Total: Double;
begin
  Total := 0;
  cds.First;
  while not cds.EOF do
  begin
    Total := Total + cdsSalary.Value;
    cds.Next;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOk], 0);
end
```

This code works, as you can see from the output in Figure 13.11, but it has some problems. One problem is that the record pointer is moved to the last record, so the previous position in the table is lost. Another is that the user interface is refreshed many times during the operation.



Williams

100

Last Name: Baldwin

First Name: Janet

Phone Ext: 2

Salary: 23300

Sum of new salaries is €1,386,202.29

FIGURE 13.11 The output of the Total program, showing the total salaries of the employees

Using Bookmarks

To avoid the two problems I just mentioned, you need to disable updates and to store the current position of the record pointer in the table and restore it at the end. You can do so using a *table bookmark*: a special variable that stores the position of a record in a database table. Delphi's traditional approach is to declare a variable of the `TBookmark` data type and initialize it while getting the current position from the table:

```
var
  Bookmark: TBookmark;
begin
  Bookmark := cds.GetBookmark;
```

At the end of the `ActionTotalExecute` method, you can restore the position and delete the bookmark with the following two statements (inside a `finally` block to ensure the pointer's memory is definitely freed):

```
cds.GotoBookmark (Bookmark);
cds.FreeBookmark (Bookmark);
```

As a better (and more up-to-date) alternative, you can use the `TDataset` class's `Bookmark` property, which refers to a bookmark that is disposed of automatically. (The property is technically implemented as an *opaque string*—a structure subject to string lifetime management—but it is not a string, so you're not supposed to look at what's inside it.) This is how you can modify the previous code:

```
var
  Bookmark: TBookmarkStr;
begin
  Bookmark := cds.Bookmark;
  ...
  cds.Bookmark := Bookmark;
```

To avoid the other side effect of the program (you see the records scrolling while the routine browses through the data), you can temporarily disable the visual controls connected with the table. The dataset has a `DisableControls` method you can call before the `while` loop starts and an `EnableControls` method you can call at the end, after the record pointer is restored.

TIP *Disabling the data-aware controls connected with a dataset during long operations not only improves the user interface (because the output is not changing constantly), but also speeds up the program considerably. The time spent updating the user interface is much greater than the time spent performing the calculations. To test this fact, try commenting out the* `DisableControls` *and* `EnableControls` *methods in the Total example and see the speed difference.*

You face some dangers from errors in reading the table data, particularly if the program is reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. To avoid this situation, you should use a `try/finally` block; to make the program 100 percent error-proof, use two nested `try/finally` blocks. Including this change and the two just discussed, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);
var
  Bookmark: TBookmarkStr;
  Total: Double;
begin
  Bookmark := Cds.Bookmark;
  try
    cds.DisableControls;
    Total := 0;
    try
      cds.First;
      while not cds.EOF do
      begin
        Total := Total + cdsSalary.Value;
        cds.Next;
      end;
    finally
      cds.EnableControls;
  end
  finally
    cds.Bookmark := Bookmark;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

NOTE *I wrote this code to show you an example of a loop to browse the contents of a dataset, but there is an alternative approach based on the use of a SQL query that returns the sum of the values of a field. When you use a SQL server, the speed advantage of a SQL call to compute the total can be significant, because you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result. There is also a better alternative when you're using a ClientDataSet, because totaling a column is one of the features provided by aggregates (discussed toward the end of this chapter). Here I discussed a generic solution, which should work for any dataset.*

Editing a Table Column

The code of the increase action is similar to the action you just saw. The `ActionIncreaseExecute` method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference: When you increase the salary, you change the data in the table. The two key statements are within the `while` loop:

```
while not cds.EOF do
begin
  cds.Edit;
  cdsSalary.Value := Round (cdsSalary.Value * SpinEdit1.Value) / 100;
  Total := Total + cdsSalary.Value;
  cds.Next;
end;
```

The first statement brings the dataset into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary, multiplying the old salary by the value of the SpinEdit component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount with the click of a button.

WARNING *Notice that the dataset enters edit mode every time the* `while` *loop is executed. This happens because in a dataset, edit operations can take place only one record at a time. You must finish the edit operation by calling* `Post` *or by moving to a different record, as in the previous code. Then, to change another record, you have to re-enter edit mode.*

Customizing a Database Grid

Unlike most other data-aware controls, which have few properties to tune, the DBGrid control has many options and is more powerful than you might think. The following sections explore some of the advanced operations you can do using a DBGrid control. The first example shows how to draw in a grid, and the second shows how to use the grid's multiple-selection feature.

Painting a DBGrid

There are many reasons you might want to customize the output of a grid. A good example is to highlight specific fields or records. Another is to provide output for fields that usually don't show up in the grid, such as BLOB, graphic, and memo fields.

To thoroughly customize the drawing of a DBGrid control, you must set its `DefaultDrawing` property to False and handle its `OnDrawColumnCell` event. If you leave the value of `DefaultDrawing` set to True, the grid will display the default output before the method is called. In that case, all you can do is add something to the default output of the grid (unless you decide to draw over it, which will take extra time and cause flickering).

The alternative approach is to call the grid's `DefaultDrawColumnCell` method, perhaps after changing the current font or restricting the output rectangle. In this last case, you can provide an extra drawing in a cell and let the grid fill the remaining area with the standard output. This is what I did in the DrawData program.

The DBGrid control in this example, which is connected to the Borland's classic Biolife table, has the following properties:

```
object DBGrid1: TDBGrid
  Align = alClient
  DataSource = DataSource1
```

```
    DefaultDrawing = False
    OnDrawColumnCell = DBGrid1DrawColumnCell
  end
```

The `OnDrawColumnCell` event handler is called once for each grid cell and has several parameters, including the rectangle corresponding to the cell, the index of the column you have to draw, the column itself (with the field, its alignment, and other subproperties), and the status of the cell. To set the color of specific cells to red, you change it in the special cases:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  // red font color if length > 100
  if (Column.Field = cdsLengthcm) and (cdsLengthcm.AsInteger > 100) then
    DBGrid1.Canvas.Font.Color := clRed;

  // default drawing
  DBGrid1.DefaultDrawDataCell (Rect, Column.Field, State);
end;
```

The next step is to draw the memo and the graphics fields. For the memo, you can implement the memo field's `OnGetText` and `OnSetText` events. The grid will even allow editing on a memo field if its `OnSetText` event is not `nil`. Here is the code for the two event handlers. I used `Trim` to remove trailing nonprinting characters that make the text appear to be empty when editing:

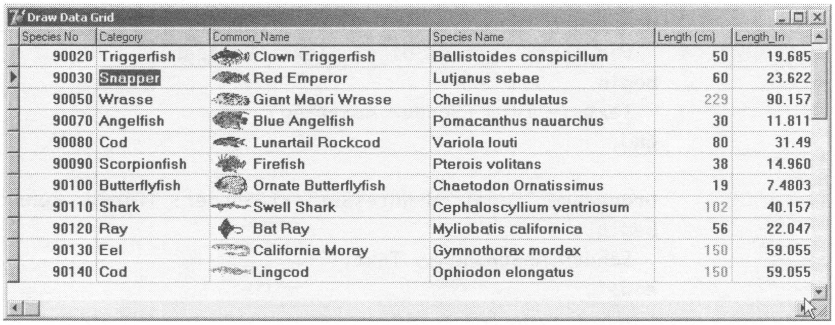
```
procedure TForm1.cdsNotesGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := Trim (Sender.AsString);
end;

procedure TForm1.cdsNotesSetText(Sender: TField; const Text: String);
begin
  Sender.AsString := Text;
end;
```

For the image, the simplest approach is to create a temporary `TBitmap` object, assign the graphics field to it, and paint the bitmap to the grid's `Canvas`. As an alternative, I removed the graphics field from the grid by setting its `Visible` property to False and added the image to the fish name, with the following extra code in the `OnDrawColumnCell` event handler:

```
var
  Picture: TPicture;
  OutRect: TRect;
  PictWidth: Integer;
begin
  // default output rectangle
  OutRect := Rect;

  if Column.Field = cdsCommon_Name then
```

```
begin
  // draw the image
  Picture := TPicture.Create;
  try
    Picture.Assign(cdsGraphic);
    PictWidth := (Rect.Bottom - Rect.Top) * 2;
    OutRect.Right := Rect.Left + PictWidth;
    DBGrid1.Canvas.StretchDraw (OutRect, Picture.Graphic);
  finally
    Picture.Free;
  end;
  // reset output rectangle, leaving space for the graphic
  OutRect := Rect;
  OutRect.Left := OutRect.Left + PictWidth;
end;

// red font color if length > 100 (omitted - see above)

// default drawing
DBGrid1.DefaultDrawDataCell (OutRect, Column.Field, State);
```

As you can see in this code, the program shows the image in a small rectangle on the left of the grid cell and then changes the output rectangle to the remaining area before activating the default drawing. You can see the effect in Figure 13.12.



Species No	Category	Common_Name	Species Name	Length (cm)	Length_In
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicillum	50	19.685
90030	Snapper	Red Emperor	Lutjanus sebae	60	23.622
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229	90.157
90070	Angelfish	Blue Angelfish	Pomacanthus nauarchus	30	11.811
90080	Cod	Lunartail Rockcod	Variola louti	80	31.49
90090	Scorpionfish	Firefish	Pterois volitans	38	14.960
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissimus	19	7.4803
90110	Shark	Swell Shark	Cephaloscyllium ventriosum	102	40.157
90120	Ray	Bat Ray	Myliobatis californica	56	22.047
90130	Eel	California Moray	Gymnothorax mordax	150	59.055
90140	Cod	Lingcod	Ophiodon elongatus	150	59.055

FIGURE 13.12

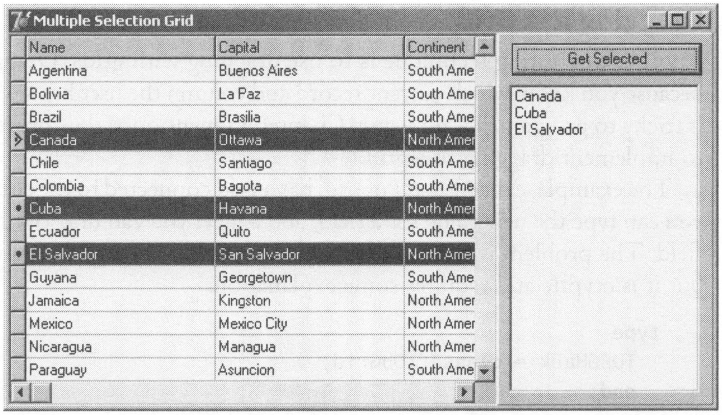
The DrawData program displays a grid that includes the text of a memo field and the ubiquitous Borland fish.

A Grid Allowing Multiple Selection

The second example of customizing the DBGrid control relates to multiple selection. You can set up the DBGrid so that a user can select multiple rows (that is, multiple records). Doing so is easy, because all you have to do is toggle the dgMultiSelect element of the grid's `Options` property. Once you select this option, a user can keep the Ctrl key pressed and click with the mouse to select multiple grid rows, with the effect shown in Figure 13.13.

Because the database table can have only one active record, the grid keeps a list of bookmarks to the selected records. This list is available in the `SelectedRows` property, which is of type `TBookmarkList`. Besides accessing the number of objects in the list with the `Count` property, you can get to each bookmark with the `Items` property, which is the default array property. Each list item is of a `TBookmarkStr` type, which represents a bookmark pointer you can assign to the table's `Bookmark` property.

FIGURE 13.13
The MltGrid example has a DBGrid control that allows the selection of multiple rows.



	Name	Capital	Continent
	Argentina	Buenos Aires	South Ame
	Bolivia	La Paz	South Ame
	Brazil	Brasilia	South Ame
▶	Canada	Ottawa	North Amer
	Chile	Santiago	South Ame
	Colombia	Bagota	South Ame
•	Cuba	Havana	North Amer
	Ecuador	Quito	South Ame
•	El Salvador	San Salvador	North Amer
	Guyana	Georgetown	South Ame
	Jamaica	Kingston	North Amer
	Mexico	Mexico City	North Amer
	Nicaragua	Managua	North Amer
	Paraguay	Asuncion	South Ame

Canada
Cuba
El Salvador

NOTE `TBookmarkStr` *is a string type for convenience, but its data should be considered opaque and volatile. You shouldn't rely on any particular structure to the data you find if you peek at a bookmark's value, and you shouldn't hold on to the data too long or store it in a separate file. Bookmark data will vary with the database driver and index configuration, and it may be rendered unusable when rows are added to or deleted from the dataset (by you or by other users of the database).*

To summarize the steps, here is the code for the MltGrid example, which is activated by clicking the button to move the Name field of the selected records to the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  BookmarkList: TBookmarkList;
  Bookmark: TBookmarkStr;
begin
  // store the current position
  Bookmark := cds.Bookmark;
  try
    // empty the list box
    ListBox1.Items.Clear;
    // get the selected rows of the grid
    BookmarkList := DbGrid1.SelectedRows;
    for I := 0 to BookmarkList.Count - 1 do
    begin
      // for each, move the table to that record
      cds.Bookmark := BookmarkList[I];
      // add the name field to the listbox
      ListBox1.Items.Add (cds.FieldByName ('Name').AsString);
    end;
  finally
    // go back to the initial record
    cds.Bookmark := Bookmark;
  end;
end;
```

Dragging to a Grid

Another interesting technique is to use dragging with grids. Dragging *from* a grid is not difficult, because you know which current record and column the user has selected. Dragging *to* a grid, however, is tricky to program. Recall that in Chapter 3 I mentioned the "protected hack"; I'll use this technique to implement dragging to a grid.

The example, called DragToGrid, has a grid connected to the country dataset, an edit box in which you can type the new value for a field, and a label you can drag over a grid cell to modify the related field. The problem is how to determine this field. The code is only a few lines, as you can see here, but it is cryptic and requires some explanation:

```
type
  TDBGHack = class (TDbGrid)
  end;

procedure TFormDrag.DBGrid1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  gc: TGridCoord;
begin
  gc := TDBGHack (DbGrid1).MouseCoord (x, y);
  if (gc.y > 0) and (gc.x > 0) then
  begin
    DbGrid1.DataSource.DataSet.MoveBy (gc.y - TDBGHack(DbGrid1).Row);
    DbGrid1.DataSource.DataSet.Edit;
    DBGrid1.Columns.Items [gc.X - 1].Field.AsString := EditDrag.Text;
  end;
  DBGrid1.SetFocus;
end;
```

The first operation determines the cell over which the mouse was released. Starting with the `x` and `y` mouse coordinates, you can call the protected `MouseCoord` method to access the row and column of the cell. Unless the drag target is the first row (usually hosting the titles) or the first column (usually hosting the indicator), the program moves the current record by the difference between the requested row (`gc.y`) and the current active row (the grid's protected `Row` property). The next step puts the dataset into edit mode, grabs the field of the target column (`Columns.Items [gc.X - 1].Field`), and changes its text.

Database Applications with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this approach is not required. When you need precise control over the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. My view is that doing so is necessary only in specific cases, because you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you better understand Delphi's default behavior.

The development of an application not based on data-aware controls can follow two different approaches: You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a more customized approach. I'll demonstrate the first technique in the NonAware example and the latter in the SendToDb example.

Mimicking Delphi Data-Aware Controls

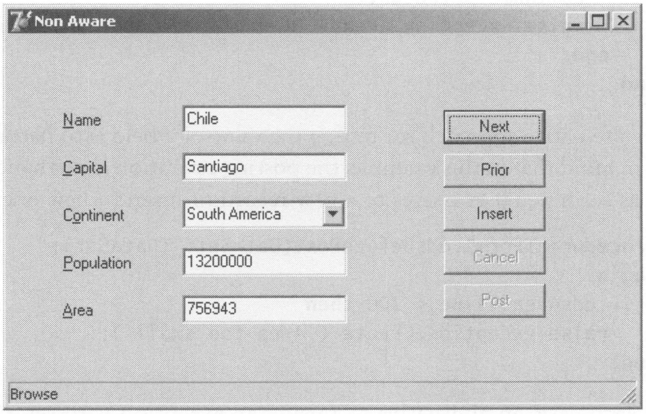
To build an application that doesn't use data-aware controls but behaves like a standard Delphi application, you can write event handlers for the operations that would be performed automatically by data-aware controls. Basically, you need to place the dataset in edit mode as the user changes the content of the visual controls and update the field objects of the dataset as the user exits from the controls, moving the focus to another element.

TIP *This approach can be handy for integrating a control that's not data aware into a standard application.*

The other element of the NonAware example is a list of buttons corresponding to some of the buttons in the DBNavigator control; these buttons are connected to five custom actions. I could not use the standard dataset actions for this example because they automatically hook to the data source associated with the control having the focus—a mechanism that fails with the example's non-data-aware edit boxes. In general, you could also hook a data Source with each of the actions' `DataSource` property, but in this specific case we don't have a data source in the example.

The program has several event handlers I haven't used for past applications using data-aware controls. First, you have to show the current record's data in the visual controls (as in Figure 13.14) by handling the `OnAfterScroll` event of the dataset component:

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);
begin
  EditName.Text := cdsName.AsString;
  EditCapital.Text := cdsCapital.AsString;
  ComboContinent.Text := cdsContinent.AsString;
  EditArea.Text := cdsArea.AsString;
  EditPopulation.Text := cdsPopulation.AsString;
end;
```



Name: Chile
Capital: Santiago
Continent: South America
Population: 13200000
Area: 756943
Browse

FIGURE 13.14 The output of the NonAware example in Browse mode. The program manually fetches the data every time the current record changes.

The control's `OnStateChange` event handler displays the table's status in a status bar control. As the user begins typing in one of the edit boxes or drops down the combo box list, the program sets the table to edit mode:

```
procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
  if not (cds.State in [dsEdit, dsInsert]) then
    cds.Edit;
end;
```

This method is connected to the `OnKeyPress` event of the five components and is similar to the `OnDropDown` event handler of the combo box. As the user leaves one of the visual controls, the `OnExit` event handler copies the data to the corresponding field, as in this case:

```
procedure TForm1.EditCapitalExit(Sender: TObject);
begin
  if (cds.State in [dsEdit, dsInsert]) then
    cdsCapital.AsString := EditCapital.Text;
end;
```

The operation takes place only if the table is in edit mode—that is, only if the user has typed in this or another control. This behavior is not ideal, because extra operations are done even if the edit box's text didn't change; however, the extra steps happen fast enough that they aren't a concern. For the first edit box, you check the text before copying it, raising an exception if the edit box is empty:

```
procedure TForm1.EditNameExit(Sender: TObject);
begin
  if (cds.State in [dsEdit, dsInsert]) then
    if EditName.Text <> '' then
      cdsName.AsString := EditName.Text
    else
    begin
      EditName.SetFocus;
      raise Exception.Create ('Undefined Country');
    end;
end;
```

An alternative approach for testing the value of a field is to handle the dataset's `BeforePost` event. Keep in mind that in this example, the posting operation is not handled by a specific button but takes place as soon as a user moves to a new record or inserts a new one:

```
procedure TForm1.cdsBeforePost(DataSet: TDataSet);
begin
  if cdsArea.Value < 100 then
    raise Exception.Create ('Area too small');
end;
```

In each case, an alternative to raising an exception is to set a default value. However, if a field has a default value, it is better to set it up front, so a user can see which value will be sent to the database. To accomplish this, you can handle the dataset's `AfterInsert` event, which is fired immediately after a new record has been created (I could have used the `OnNewRecord` event, as well):

```
procedure TForm1.cdsAfterInsert(DataSet: TDataSet);
begin
  cdsContinent.Value := 'Asia';
end;
```

Sending Requests to the Database

You can further customize your application's user interface if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This approach allows you complete freedom, although it might cause some side effects (such as limited ability to handle concurrency, which I'll discuss in Chapter 14).

For this new example, I replaced the first edit box with another combo box and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom buttons that get the data from the database and send an update to it. Again, this example has no DataSource component.

The `GetData` method, connected to the corresponding button, gets the fields corresponding to the record indicated in the first combo box:

```
procedure TForm1.GetData;
begin
  cds.Locate ('Name', ComboName.Text, [loCaseInsensitive]);
  ComboName.Text := cdsName.AsString;
  EditCapital.Text := cdsCapital.AsString;
  ComboContinent.Text := cdsContinent.AsString;
  EditArea.Text := cdsArea.AsString;
  EditPopulation.Text := cdsPopulation.AsString;
end;
```

This method is called whenever the user clicks the button, selects an item in the combo box, or presses the Enter key while in the combo box:

```
procedure TForm1.ComboNameClick(Sender: TObject);
begin
  GetData;
end;

procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
    GetData;
end;
```

To make this example work smoothly, at startup the combo box is filled with the names of all the countries in the table:

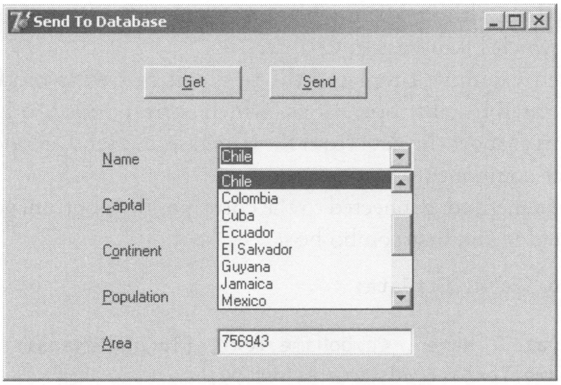
```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
  // fill the list of names
  cds.Open;
  while not cds.Eof do
  begin
    ComboName.Items.Add (cdsName.AsString);
    cds.Next;
  end;
end;
```

With this approach, the combo box becomes a sort of selector for the record, as you can see in Figure 13.15. Thanks to this selection, the program doesn't need navigational buttons.

FIGURE 13.15

In the SendToDb example, you can use a combo box to select the record you want to see.



Name: Chile

Area: 756943

The user can also change the values of the controls and click the Send button. The code to be executed depends on whether the operation is an update or an insert. You can determine this by looking at the name (although with this code, a wrong name can no longer be modified):

```
procedure TForm1.SendData;
begin
  // raise an exception if there is no name
  if ComboName.Text = '' then
    raise Exception.Create ('Insert the name');

  // check if the record is already in the table
  if cds.Locate ('Name', ComboName.Text, [loCaseInsensitive]) then
  begin
    // modify found record
    cds.Edit;
    cdsCapital.AsString := EditCapital.Text;
    cdsContinent.AsString := ComboContinent.Text;
    cdsArea.AsString := EditArea.Text;
    cdsPopulation.AsString := EditPopulation.Text;
    cds.Post;
  end
```

```
else
begin
  // insert new record
  cds.InsertRecord ([ComboName.Text, EditCapital.Text,
    ComboContinent.Text, EditArea.Text, EditPopulation.Text]);
  // add to list
  ComboName.Items.Add (ComboName.Text)
end;
```

Before sending the data to the table, you can do any sort of validation test on the values. In this case, it doesn't make sense to handle the events of the database components, because you have full control over when the update or insert operation is performed.

Grouping and Aggregates

You've already seen that a ClientDataSet can have an index different from the order in which the data is stored in the file. Once you define an index, you can group the data by that index. In practice, a *group* is defined as a list of consecutive records (according to the index) for which the value of the indexed field doesn't change. For example, if you have an index by state, all the addresses within that state will fall in the group.

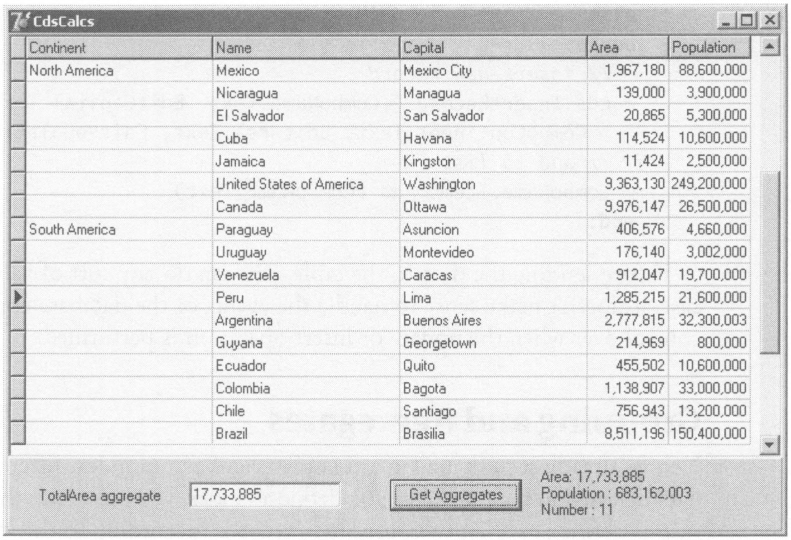
Grouping

The CdsCalcs example has a ClientDataSet component that extracts its data from the familiar Country dataset. The group is obtained, along with the definition of an index, by specifying a grouping level for the index:

```
object ClientDataSet1: TClientDataSet
  IndexDefs = <
    item
      Name = 'ClientDataSet1Index1'
      Fields = 'Continent'
      GroupingLevel = 1
    end>
  IndexName = 'ClientDataSet1Index1'
```

When a group is active, you can make it obvious to the user by displaying the grouping structure in the DBGrid, as shown in Figure 13.16. All you have to do is handle the `OnGetText` event for the grouped field (the Continent field in the example) and show the text only if the record is the first in the group:

```
procedure TForm1.ClientDataSet1ContinentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if gbFirst in ClientDataSet1.GetGroupState (1) then
    Text := Sender.AsString
  else
    Text := '';
end;
```



Continent	Name	Capital	Area	Population
North America	Mexico	Mexico City	1,967,180	88,600,000
	Nicaragua	Managua	139,000	3,900,000
	El Salvador	San Salvador	20,865	5,300,000
	Cuba	Havana	114,524	10,600,000
	Jamaica	Kingston	11,424	2,500,000
	United States of America	Washington	9,363,130	249,200,000
	Canada	Ottawa	9,976,147	26,500,000
South America	Paraguay	Asuncion	406,576	4,660,000
	Uruguay	Montevideo	176,140	3,002,000
	Venezuela	Caracas	912,047	19,700,000
	Peru	Lima	1,285,215	21,600,000
	Argentina	Buenos Aires	2,777,815	32,300,003
	Guyana	Georgetown	214,969	800,000
	Ecuador	Quito	455,502	10,600,000
	Colombia	Bagota	1,138,907	33,000,000
	Chile	Santiago	756,943	13,200,000
	Brazil	Brasilia	8,511,196	150,400,000

TotalArea aggregate: 17,733,885

```
Area: 17,733,885
Population : 683,162,003
Number : 11
```

FIGURE 13.16
The CdsCalcs example demonstrates that by writing a little code, you can have the DBGrid control visually show the grouping defined in the ClientDataSet.

Defining Aggregates

Another feature of the ClientDataSet component is support for aggregates. An *aggregate* is a calculated value based on multiple records, such as the sum or average value of a field for the entire table or a group of records (defined with the grouping logic I just discussed). Aggregates are *maintained*; that is, they are recalculated immediately if one of the records changes. For example, the total of an invoice can be maintained automatically while the user types in the invoice items.

NOTE *Aggregates are maintained incrementally, not by recalculating all the values every time one value changes. Aggregate updates take advantage of the deltas tracked by the ClientDataSet. For example, to update a sum when a field is changed, the ClientDataSet subtracts the old value from the aggregate and adds the new value. Only two calculations are needed, even if there are thousands of rows in that aggregate group. For this reason, aggregate updates are instantaneous.*

There are two ways to define aggregates. You can use the `Aggregates` property of the ClientDataSet, which is a collection; or you can define aggregate fields using the Fields Editor. In both cases, you define the aggregate expression, give it a name, and connect it to an index and a grouping level (unless you want to apply it to the entire table). Here is the `Aggregates` collection of the CdsCalcs example:

```
object ClientDataSet1: TClientDataSet
  Aggregates = <
    item
      Active = True
      AggregateName = 'Count'
      Expression = 'COUNT (NAME)'
      GroupingLevel = 1
      IndexName = 'ClientDataSet1Index1'
      Visible = False
    end
    item
      Active = True
      AggregateName = 'TotalPopulation'
      Expression = 'SUM (POPULATION)'
      Visible = False
    end>
  AggregatesActive = True
```

Notice in the last line of the previous code snippet that you must activate the support for aggregates, in addition to activating each specific aggregate you want to use. Disabling aggregates is important, because having too many of them can slow down a program.

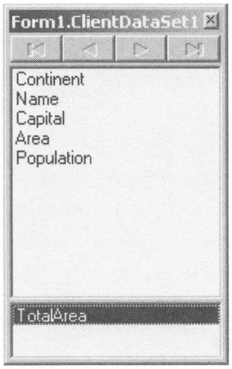
The alternative approach, as I mentioned, is to use the Fields Editor, select the New Field command from its shortcut menu, and choose the Aggregate option (available, along with the InternalCalc option, only in a ClientDataSet). This is the definition of an aggregate field:

```
object ClientDataSet1: TClientDataSet
  object ClientDataSet1TotalArea: TAggregateField
    FieldName = 'TotalArea'
    ReadOnly = True
    Visible = True
    Active = True
    DisplayFormat = '###,###,###'
    Expression = 'SUM(AREA)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
  end
```

The aggregate fields are displayed in the Fields Editor in a separate group, as you can see in Figure 13.17. The advantage of using an aggregate field, compared to a plain aggregate, is that you can define the display format and hook the field directly to a data-aware control, such as a DBEdit in the CdsCalcs example. Because the aggregate is connected to a group, as soon as you select a record from a different group, the output is automatically updated. Also, if you change the data, the total immediately shows the new value.

FIGURE 13.17

The bottom portion of a ClientDataSet's Fields Editor displays aggregate fields.



To use plain aggregates, you have to write a little code, as in the following example (notice that the `Value` of the aggregate is a variant):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption :=
    'Area: ' + ClientDataSet1TotalArea.DisplayText + #13'Population : '
    + FormatFloat ('###,###,###', ClientDataSet1.Aggregates [1].Value) +
    #13'Number : ' + IntToStr (ClientDataSet1.Aggregates [0].Value);
end;
```

Master/Detail Structures

Often, you need to relate tables that have a one-to-many relationship. This means that for a single record in the master table, there are many detailed records in a secondary table. A classic example is an invoice and the items of the invoice; another is a list of customers and the orders each customer has placed.

These are common situations in database programming, and Delphi provides explicit support with the master/detail structure. The `TDataSet` class has a `DataSource` property for setting up a master data source. This property is used in a detail dataset to hook to the current record of the master dataset, in combination with the `MasterFields` property.

Master/Detail with ClientDataSets

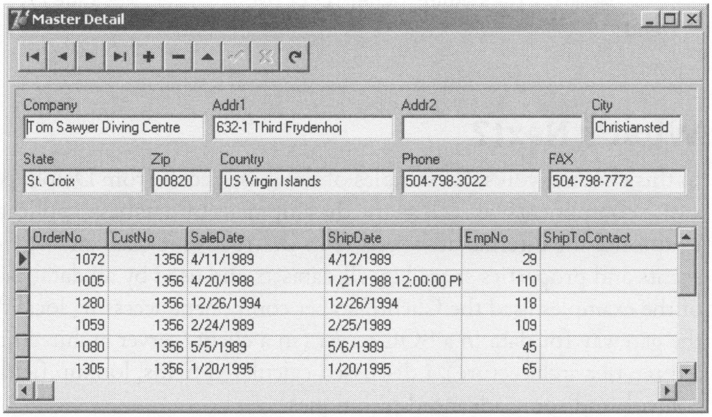
The MastDet example uses the customer and orders sample datasets. I added a data source component for each dataset, and for the secondary dataset I assigned the `DataSource` property to the data source connected to the first dataset. Finally, I related the secondary table to a field of the main table, using the `MasterFields` property's special editor. I did all this using a data module, as discussed in the earlier sidebar "A Data Module for Data-Access Components."

The following is the complete listing (but without the irrelevant positional properties) of the data module used by the MastDet program:

```
object DataModule1: TDataModule1
  OnCreate = DataModule1Create
  object dsCust: TDataSource
    DataSet = cdsCustomers
  end
  object dsOrd: TDataSource
    DataSet = cdsOrders
  end
  object cdsOrders: TClientDataSet
    FileName = 'orders.cds'
    IndexFieldNames = 'CustNo'
    MasterFields = 'CustNo'
    MasterSource = dsCust
  end
  object cdsCustomers: TClientDataSet
    FileName = 'customer.cds'
  end
end
```

In Figure 13.18, you can see an example of the MastDet program's main form at run time. I placed data-aware controls related to the master table in the upper portion of the form, and I placed a grid connected with the detail table in the lower portion. This way, for every master record, you immediately see the list of connected detail records—in this case, all orders placed by the current client. Each time you select a new customer, the grid below the master record displays only the orders pertaining to that customer.

FIGURE 13.18
The MastDet example at run time



Company	Addr1	Addr2	City
Tom Sawyer Diving Centre	632-1 Third Frydenhoj		Christiansted

State	Zip	Country	Phone	FAX
St. Croix	00820	US Virgin Islands	504-798-3022	504-798-7772

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact
1072	1356	4/11/1989	4/12/1989	29	
1005	1356	4/20/1988	1/21/1988 12:00:00 PI	110	
1280	1356	12/26/1994	12/26/1994	118	
1059	1356	2/24/1989	2/25/1989	109	
1080	1356	5/5/1989	5/6/1989	45	
1305	1356	1/20/1995	1/20/1995	65	

Handling Database Errors

Another important element of database programming is handling database errors in custom ways. Of course, you can let Delphi show an exception message each time a database error occurs, but you might want to try to correct the errors or show more details. You can use three approaches to handle database-related errors:

- Wrap a `try/except` block around risky database operations. This is not possible when the operation is generated by interaction with a data-aware control.
- Install a handler for the `OnException` event of the global `Application` object.
- Handle specific dataset events related to errors, such as `OnPostError`, `OnEditError`, `OnDeleteError`, and `OnUpdateError`.

Although most of the exception classes in Delphi deliver an error message, database exceptions often include error codes, native SQL server error codes and messages, and the like. The ClientDataSet adds only an error code to its exception class, `EDBClient`. Showing how to handle it, as I'll do next, will provide you with a guideline for other cases.

As an example, I built a database program that shows the details of the errors in a memo component (errors are automatically generated when the user clicks the program buttons). To handle all the errors, the DBError example installs a handler for the `OnException` event of the `Application` global object. The event handler logs some information in a memo showing the details of the database error if it is an `EDBClient`:

```
procedure TForm1.ApplicationError (Sender: TObject; E: Exception);
begin
  if E is EDBClient then
```

```
    begin
      Memo1.Lines.Add('Error: ' + (E.Message));
      Memo1.Lines.Add('    Error Code: ' +
        IntToStr(EDBClient (E).ErrorCode));
    end
    else
      Memo1.Lines.Add('Generic Error: ' + (E.Message));
  end;
```

What's Next?

In this chapter, you saw examples of database access from Delphi programs. I covered the basic data-aware components, as well as the development of database applications based on standard controls. I explored the internal architecture of the `TDataSet` class and of field objects and discussed many events and properties shared by all datasets and used by all database applications. Even though most of the examples used the ClientDataSet component accessing local data, the component is often also the gateway for data in a SQL server (in a client/server architecture) or in a remote application (in a three-tier architecture). I discussed calculated fields, lookup fields, customizations of the DBGrid control, and some advanced techniques.

I haven't delved into the database and data-access side of the picture, which depends on the type of database engine and server you're using. Chapter 14 will focus on this topic, with an in-depth overview of client/server development using the dbExpress library provided by Borland. I'll also cover InterBase and the IBX component, giving some elements of a real-world application.

Following chapters will continue to explore the database side of Delphi programming, discussing ADO connectivity and components, Borland's three-tier architecture (DataSnap, formerly MIDAS), the development of data-aware controls and custom dataset components, and reporting technologies.

Chapter 14

Client/Server with dbExpress

In THE LAST CHAPTER, we examined Delphi's support for database programming, using local files (particularly using the ClientDataSet component, or MyBase) in most of the examples but not focusing on any specific database technology. This chapter moves on to the use of SQL server databases, focusing on client/server development with the BDE and the new dbExpress technology. A single chapter cannot cover this complex topic in detail, so I'll introduce it from the perspective of the Delphi developer and add some tips and hints.

For the examples I'll use InterBase, because this Borland RDBMS (relational database management system), or SQL server, is included in the Professional and higher editions of Delphi; in addition, it is a free and open-source server (although not in all of its versions, as discussed later). I'll discuss InterBase from the Delphi perspective, without delving into its internal architecture. A lot of the information presented also applies to other SQL servers, so even if you've decided not to use InterBase, you may still find it valuable.

This chapter covers the following:

- Overview of client/server
- Elements of database design
- Introducing InterBase
- Server-side programming: views, stored procedures, and triggers
- The dbExpress library
- Caching with the ClientDataSet component
- The InterBase Express (IBX) components

The Client/Server Architecture

The database applications presented in previous chapters used native components to access data stored in files on the local machine and loaded the entire file in memory. This is an extreme approach.

More traditionally, the file is read record by record so that multiple applications can access it at the same time, provided write synchronization mechanisms are used.

When the data is on a remote server, copying an entire table in memory for processing it is time- and bandwidth-consuming, and often also useless. As an example, consider taking a table like EMPLOYEE (part of the InterBase sample database, which ships with Delphi), adding thousands of records to it, and placing it on a networked computer working as a file server. If you want to know the highest salary paid by the company, you can open a dbExpress table component (`EmpTable`) or a query selecting all the records, and run this code:

```
EmpTable.Open;
EmpTable.First;
MaxSalary := 0;
while not EmpTable.Eof do
begin
  if EmpTable.FieldByName ('Salary').AsCurrency > MaxSalary then
    MaxSalary := EmpTable.FieldByName ('Salary').AsCurrency;
  EmpTable.Next;
end;
```

The effect of this approach is to move all the data of the table from the networked computer to the local machine—an operation that might take minutes. In this case, the proper approach is to let the SQL server compute the result directly, fetching only this single piece of information. You can do so using a SQL statement like this:

```
select Max(Salary) from Employee
```

NOTE *The previous two code excerpts are part of the GetMax example, which includes code to time the two approaches. Using the Table component on the small Employee table takes about 10 times longer than using the query, even if the InterBase server is installed on the computer running the program.*

To store a large amount of data on a central computer and avoid moving the data to client computers for processing, the only solution is to let the central computer manipulate the data and send back to the client only a limited amount of information. This is the foundation of client/server programming.

In general, you'll use an existing program on the server (an RDBMS) and write a custom client application that connects to it. Sometimes, however, you may want to write both a custom client and a custom server, as in three-tier applications. Delphi support for this type of program—which has been called the Middle-tier Distributed Application Services (MIDAS) architecture and is now dubbed DataSnap—is covered in Chapter 16, "Multitier DataSnap Applications."

The *upsizing* of an application—that is, the transfer of data from local files to a SQL server database engine—is generally done for performance reasons and to allow for larger amounts of data. Going back to the previous example, in a client/server environment, the query used to select the maximum salary would be computed by the RDBMS, which would send back to the client computer only the final result—a single number. With a powerful server computer (such as a multiprocessor Sun SparcStation), the total time required to compute the result might be minimal.

However, there are other reasons to choose a client/server architecture. Such an architecture:

- Helps you manage a larger amount of data, because you don't want to store hundreds of megabytes in a local file.
- Supports the need for concurrent access to the data by multiple users at the same time. SQL server databases generally use *optimistic locking*, an approach that allows multiple users to work on the same data and delays the concurrency control until users send back updates.
- Provides data integrity, transaction control, security, access control, backup support, and the like.
- Supports programmability—the possibility of running part of the code (stored procedures, triggers, table views, and other techniques) on the server, thereby reducing the network traffic and the workload of the client computers.

Having said this, we can begin focusing on particular techniques useful for client/server programming. The general goal is to distribute the workload properly between the client and the server and reduce the network bandwidth required to move information back and forth.

The foundation of this approach is good database design, which involves both table structure and appropriate data validation and constraints, or business rules. Enforcing the validation of the data on the server is important, because the integrity of the database is one of the key aims of any program. However, the client side should include data validation as well, to improve the user interface and make the input and the processing of the data more user-friendly. It makes little sense to let the user enter invalid data and then receive an error message from the server, when you can prevent the wrong input in the first place.

Elements of Database Design

Although this is a book about Delphi programming, not databases, I feel it's important to discuss a few elements of good (and modern) database design. The reason is simple: If your database design is incorrect or convoluted, you'll either have to write terribly complex SQL statements and server-side code, or write a lot of Delphi code to access your data, possibly even fighting against the design of the `TDataSet` class.

Entities and Relations

The classic relational database design approach, based on the entity-relation (E-R) model, involves having one table for every entity you need to represent in your database, with one field for each data element you need plus one field for every one-to-one or one-to-many relation to another entity (or table). For many-to-many relations, you need a separate table.

As an example of a one-to-one relation, consider a table representing a university course. It will have a field for each relevant data element (name and description, room where the course is held, and so on) plus a single field indicating the teacher. The teacher data really should not be stored within the course data, but in a separate table, because it may be referenced from elsewhere.

The schedule for each course can include an undefined number of hours on different days, so they cannot be added in the same table describing the course. Instead, this information must be placed in a separate table that includes all the schedules, with a field referring to the class each schedule is for. In a one-to-many relation like this, *many* records of the schedule table point to the same *one* record in the course table.

A more complex situation is required to store information about which student is taking which class. Students cannot be listed directly in the course table, because their number is not fixed, and the classes cannot be stored in the student's data for the same reason. In a similar many-to-many relation, the only approach is to create an extra table representing the relation—it lists references to students and courses.

NORMALIZATION RULES

The classic design principles include a series of so-called *normalization* rules. The goal of these rules is to avoid duplicating data in your database (not only to save space, but mainly to avoid ending up with incongruous data). For example, you don't repeat all the customer details in each order, but refer to a separate customer entity. This way you save memory, and when a customer's details change (for example, because of a change of address), all of the customer's orders reflect the new data. Other tables that relate to the same customer will be automatically updated as well.

Normalization rules imply using codes for commonly repeated values. For example, suppose you have a few different shipment options. Rather than include a string-based description for these options within the orders table, you can use a short numeric code that's mapped to a description in a separate lookup table.

The previous rule, which should not be taken to the extreme, helps you avoid having to join a large number of tables for every query. You can either account for some de-normalization (leaving a short shipment description within the orders table) or use the client program to provide the description, again ending up with a formally incorrect database design. This last option is practical only when you use a single development environment (let's say, Delphi) to access this database.

From Primary Keys to OIDs

In a relational database, records are identified not by a physical position (as in Paradox and other local databases) but by the data within the record. Typically, you don't need the data from every field to identify a record, but only a subset of the data, forming the *primary key*. If the fields that are part of the primary key must identify an individual record, their value must be different for each possible record of the table.

NOTE *Many database servers add internal record identifiers to tables, but they do so only for internal optimization; this process has little to do with the logical design of a relational database. These internal identifiers work differently in different SQL servers and may change among versions, so you shouldn't rely on them.*

Early incarnations of relational theory dictated the use of *logical keys*, which means selecting one or more fields that indicate an entity without risk of confusion. This is often easier to say than to accomplish. For example, company names are not generally unique, and even the company name and its location don't provide a complete guarantee of uniqueness. Moreover, if a company changes its name (not an unlikely event, as Borland can teach us) or its location, and you have references to the company in other tables, you must change all those references as well and risk ending up with dangling references.

For this reason, and also for efficiency (using strings for references implies using a lot of space in secondary tables, where references often occur), logical keys have been phased out in favor of physical or surrogate keys:

Physical Key A single field that identifies an element in a unique way. For example, each person in the U.S. has a Social Security Number (SSN), but almost every country has a tax ID or other government-assigned number that identifies each person. The same is typically true for companies. Although these ID numbers are guaranteed to be unique, they can change depending on the country (creating troubles for the database of a company that sells goods abroad) or within a single country (to account for new tax laws). They are also often inefficient, because they can be quite large (Italy, for example, uses a 16-character code—letters and numbers—to identify people).

Surrogate Key A number identifying a record, in the form of a client code, order number, and so on. Surrogate keys are commonly used in database design. However, in many cases, they end up being *logical identifiers*, with client codes showing up all over the place (not a great idea).

WARNING *The situation becomes particularly troublesome when surrogate keys also have a meaning and must follow specific rules. For example, companies must number invoices with unique and consecutive numbers, without leaving holes in the numbering sequence. This situation is extremely complex to handle programmatically, if you consider that only the database can determine these unique consecutive numbers when you send it new data. At the same time, you need to identify the record before you send it to the database—otherwise you won't be able to fetch it again. Practical examples of how to solve this situation are discussed in Chapter 15, "Working with ADO."*

OIDS TO THE EXTREME

An extension to the use of surrogate keys is the use of a unique Object Identifier (OID). An OID is either a number or a string with a sequence of numbers and digits; it's added to each record of each table representing an entity (and sometimes to records of tables representing relations). Unlike client codes, invoice numbers, SSNs, or purchase order numbers, OIDs are random: They have no sequencing rule and are never visible to the end user. This means you can use surrogate keys (if your company is used to them) along with OIDs, but all the external references to the table will be based on OIDs.

Another common rule suggested by the promoters of this approach (which is part of the theories supporting object-relational mapping) is the use of system-wide unique identifiers. If you have a table of client companies and a table of employees, you may wonder why you should use a unique ID for such diverse data. The reason is that you'll be able to sell goods to an employee without having to duplicate the employee information in the customer table—you can refer to the employee in your order and invoice. An order is placed by someone identified by an OID, and this OID can refer to many different tables.

Using OIDs and object-relational mapping is an advanced element of the design of Delphi database applications. I suggest that you investigate this topic before embracing medium or large Delphi projects because the benefit can be relevant (after some investment in studying this approach and building some basic support code).

EXTERNAL KEYS AND REFERENTIAL INTEGRITY

The keys identifying a record (whatever their type) can be used as external keys in other tables—for example, to represent the various types of relations discussed earlier. All SQL servers can verify these external references, so you cannot refer to a nonexistent record in another table. These referential integrity constraints are expressed when you create a table.

Besides not being allowed to add references to nonexistent records, you're generally prevented from deleting a record if external references to it exist. Some SQL servers go one step further: As you delete a record, instead of denying the operation, they can automatically delete all records that refer to it from other tables.

More Constraints

In addition to the uniqueness of primary keys and the referential constraints, you can generally use the database to impose more validity rules on the data. You can ask for specific columns (such as those referring to a tax ID or a purchase order number) to include only unique values. You can impose uniqueness on the values of multiple columns—for example, to indicate that you cannot hold two classes in the same room at the same time.

In general, simple rules can be expressed to impose constraints on a table, whereas more complex rules generally imply the execution of stored procedures activated by triggers (every time the data changes, for instance, or there is new data).

Again, there is much more to proper database design, but the elements discussed in this section can provide you with a starting point or a good refresher.

NOTE *For more information about SQL's Data Definition Language and Data Manipulation Language, see the chapter "Essential SQL" in the electronic book described in Appendix C, "Free Companion Books on Delphi."*

Unidirectional Cursors

In local databases, tables are sequential files whose order either is the physical order or is defined by an index. By contrast, SQL servers work on logical sets of data that aren't related to a physical order. A *relational* database server handles data according to the relational model: a mathematical model based on set theory.

For this discussion, it's important for you to know that in a relational database, the records (sometimes called *tuples*) of a table are identified not by position but exclusively through a primary key, based on one or more fields. Once you've obtained a set of records, the server adds to each of them a reference to the following record; thus you can move quickly from a record to the following one, but moving back to the previous record is extremely slow. For this reason, it is common to say that an RDBMS uses a *unidirectional* cursor. Connecting such a table or query to a DBGrid control is practically impossible, because doing so would make browsing the grid backward terribly slow.

Some database engines keep the data already retrieved in a cache, to support full *bidirectional* navigation on it. In the Delphi architecture, this role can be played by the ClientDataSet component or another caching dataset. You'll see this process in more detail later, when we focus on dbExpress and the SQLDataset component.

NOTE *The case of a DBGrid used to browse an entire table is common in local programs but should generally be avoided in a client/server environment. It's better to filter out only part of the records and only the fields you are interested in. If you need to see a list of names, return all those starting with the letter* A, *then those with* B, *and so on, or ask the user for the initial letter of the name.*

If proceeding backward might result in problems, keep in mind that jumping to the last record of a table is even worse; usually this operation implies fetching all the records! A similar situation applies to the `RecordCount` property of datasets. Computing the number of records often implies moving them all to the client computer. For this reason, the thumb of the DBGrid's vertical scrollbar works for a local table but not for a remote table. If you need to know the number of records, run a separate query to let the server (and not the client) compute it. For example, you can see how many records will be selected from the EMPLOYEE table if you are interested in those records having a salary field higher than 50,000:

```
select count(*)
from Employee
where Salary > 50000
```

TIP *Using the SQL instruction* `count(*)` *is a handy way to compute the number of records returned by a query. Instead of the * wildcard, you could use the name of a specific field, as in* `count(First_Name)`*, possibly combined with either* `distinct` *or* `all`*, to count only records with different values for the field or all the records having a non-*`null` *value.*

Introducing InterBase

Although it has a limited market share, InterBase is a powerful RDBMS. In this section, I'll introduce the key technical features of InterBase without getting into too much detail (because this is a book about Delphi programming). Unfortunately, little is currently published about InterBase. Most of the available material is either in the documentation that accompanies the product or on a few websites devoted to it (your starting points for a search can be `www.borland.com/interbase` and `www.ibphoenix.com`).

InterBase was built from the beginning with a modern and robust architecture. Its original author, Jim Starkey, invented an architecture for handling concurrency and transactions without imposing physical locks on portions of the tables, something other well-known database servers can barely do even today. The InterBase architecture is called Multi-Generational Architecture (MGA); it handles concurrent access to the same data by multiple users, who can modify records without affecting what other concurrent users see in the database.

This approach naturally maps to the *Repeatable Read* transaction isolation mode, in which a user within a transaction keeps seeing the same data regardless of changes made and committed by other users. Technically, the server handles this situation by maintaining a different version of each accessed record for each open transaction. Even though this approach (also called *versioning*) can lead to larger memory consumption, it avoids most physical locks on the tables and makes the system much more robust in case of a crash. MGA also pushes toward a clear programming model—Repeatable Read—which other well-known SQL servers don't support without losing most of their performance.

In addition to the MGA at the heart of InterBase, the server has many other technical advantages:

- A limited footprint, which makes InterBase the ideal candidate for running directly on client computers, including portables. The disk space required by InterBase for a minimal installation is well below 10 MB, and its memory requirements are also incredibly limited.
- Good performance on large amounts of data.
- Availability on many different platforms (including 32-bit Windows, Solaris, and Linux), with totally compatible versions. Thus the server is scalable from very small to huge systems without notable differences.
- A very good track record, because InterBase has been in use for 15 years with few problems.
- A language complaint with the ANSI SQL standard.

- Advanced programming capabilities, including positional triggers, selectable stored procedures, updateable views, exceptions, events, generators, and more.
- Simple installation and management, with limited administrative headaches.

A SHORT HISTORY OF INTERBASE

Jim Starkey wrote InterBase for his Groton Database Systems company (hence the `.gds` extension still used for InterBase files). The company was later bought by Ashton-Tate, which was then acquired by Borland. Borland handled InterBase directly for a while and then created an InterBase subsidiary, which was later re-absorbed into the parent company.

Beginning with Delphi 1, an evaluation copy of InterBase has been distributed with the development tool, spreading the database server among developers. Although it doesn't have a large piece of the RDBMS market, which is dominated by a handful of players, InterBase has been chosen by a few relevant organizations, from Ericsson to the U.S. Department of Defense, from stock exchanges to home banking systems.

More recent events include the announcement of InterBase 6 as an open-source database (December 1999), the effective release of source code to the community (July 2000), and the release of the officially certified version of InterBase 6 by Borland (March 2001). Between these events came announcements of the spin-off of a separate company to run the consulting and support business in addition to the open-source database. A group of former InterBase developers and managers (who had left Borland) formed IBPhoenix (`www.ibphoenix.com`) with the plan of supporting InterBase users.

At the same time, independent groups of InterBase experts started the Firebird open-source project to further extend InterBase. The project is hosted on SourceForge at the address `sourceforge.net/projects/firebird/`. For some time, SourceForge also hosted a Borland open-source project, but later the company announced it would continue to support only its proprietary version, dropping its open-source effort. So, the picture is now clearer. If you want a version with a traditional license (costing a fraction of most competing professional SQL servers), stick with Borland; but if you prefer an open-source, totally free model, go with the Firebird project (and eventually buy professional support from IBPhoenix).

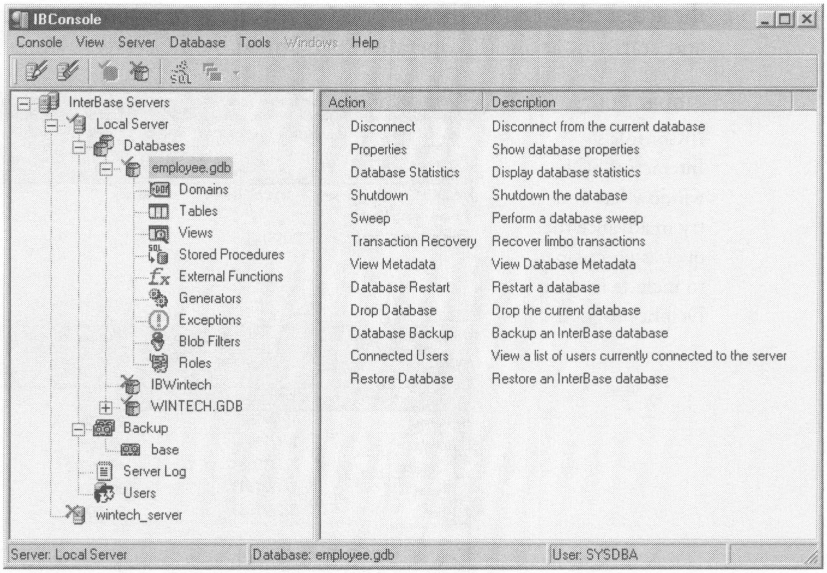
Using IBConsole

In past versions of InterBase, you could use two primary tools to interact directly with the program: the Server Manager application, which could be used to administer both a local and a remote server; and Windows Interactive SQL (WISQL). Version 6 includes a much more powerful front-end application, called IBConsole. This full-fledged Windows program (built with Delphi) allows you to administer, configure, test, and query an InterBase server, whether local or remote.

IBConsole is a simple and complete system for managing InterBase servers and their databases. You can use it to look into the details of the database structure, modify it, query the data (which can be useful to develop the queries you want to embed in your program), back up and restore the database, and perform any other administrative tasks.

As you can see in Figure 14.1, IBConsole allows you to manage multiple servers and databases, all listed in a single configuration tree. You can ask for general information about the database and list its entities (tables, domains, stored procedures, triggers, and everything else), accessing the details of each. You can also create new databases and configure them, back up the files, update the definitions, check what's going on and who is currently connected, and so on.

FIGURE 14.1
IBConsole lets you manage, from a single computer, InterBase databases hosted by multiple servers.

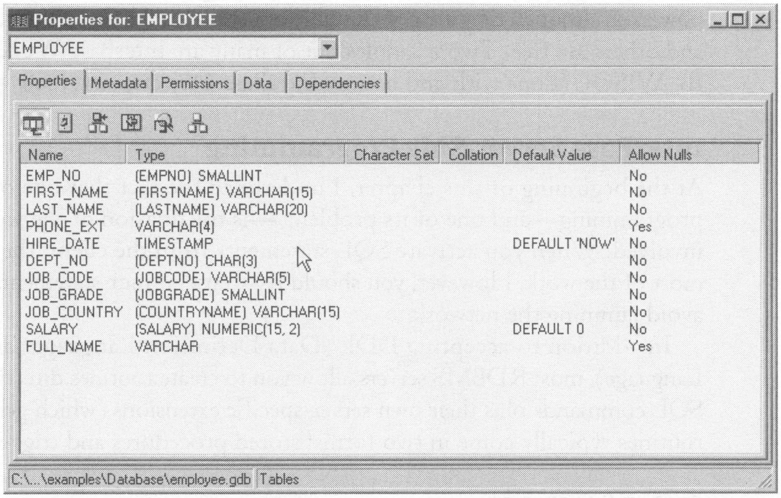

IBConsole
Console View Server Database Tools Windows Help
Server: Local Server
Database: employee.gdb
User: SYSDBA

- InterBase Servers
  - Local Server
    - Databases
      - employee.gdb
        - Domains
        - Tables
        - Views
        - Stored Procedures
        - External Functions
        - Generators
        - Exceptions
        - Blob Filters
        - Roles
      - IBWintech
      - WINTECH.GDB
    - Backup
      - base
    - Server Log
    - Users
  - wintech_server

Action	Description
Disconnect	Disconnect from the current database
Properties	Show database properties
Database Statistics	Display database statistics
Shutdown	Shutdown the database
Sweep	Perform a database sweep
Transaction Recovery	Recover limbo transactions
View Metadata	View Database Metadata
Database Restart	Restart a database
Drop Database	Drop the current database
Database Backup	Backup an InterBase database
Connected Users	View a list of users currently connected to the server
Restore Database	Restore an InterBase database

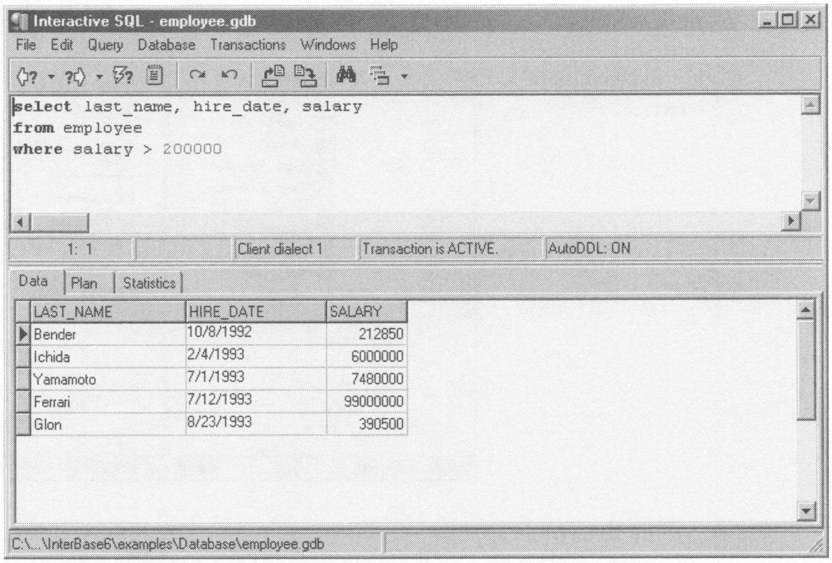
The IBConsole application allows you to open multiple windows to look at detailed information, such as the tables window shown in Figure 14.2. In this window, you can see lists of the key properties of each table (columns, triggers, constraints, and indexes), see the raw metadata (the SQL definition of the table), access permissions, look at the data, modify the data, and study the table's dependencies. Similar windows are available for each of the other entities you can define in a database.

FIGURE 14.2
IBConsole can open separate windows to show you the details of each entity—in this case, a table.


Properties for: EMPLOYEE
EMPLOYEE
Properties
Metadata
Permissions
Data
Dependencies
C:\...\examples\Database\employee.gdb
Tables

Name	Type	Character Set	Collation	Default Value	Allow Nulls
EMP_NO	(EMPNO) SMALLINT				No
FIRST_NAME	(FIRSTNAME) VARCHAR(15)				No
LAST_NAME	(LASTNAME) VARCHAR(20)				No
PHONE_EXT	VARCHAR(4)				Yes
HIRE_DATE	TIMESTAMP			DEFAULT 'NOW'	No
DEPT_NO	(DEPTNO) CHAR(3)				No
JOB_CODE	(JOBCODE) VARCHAR(5)				No
JOB_GRADE	(JOBGRADE) SMALLINT				No
JOB_COUNTRY	(COUNTRYNAME) VARCHAR(15)				No
SALARY	(SALARY) NUMERIC(15, 2)			DEFAULT 0	No
FULL_NAME	VARCHAR				Yes

IBConsole embeds an improved version of the original Windows Interactive SQL application (see Figure 14.3). You can type a SQL statement in the upper portion of the window (without any help from the tool, unfortunately) and then execute the SQL query. As a result, you'll see the data, but also the access plan used by the database (which an expert can use to determine the efficiency of the query) and statistics about the operation performed by the server.


Interactive SQL - employee.gdb
File Edit Query Database Transactions Windows Help
Data
Plan
Statistics

```
select last_name, hire_date, salary
from employee
where salary > 200000
```

1: 1 | Client dialect 1 | Transaction is ACTIVE. | AutoDDL: ON

LAST_NAME	HIRE_DATE	SALARY
Bender	10/8/1992	212850
Ichida	2/4/1993	6000000
Yamamoto	7/1/1993	7480000
Ferrari	7/12/1993	99000000
Glon	8/23/1993	390500

C:\...\InterBase6\examples\Database\employee.gdb

FIGURE 14.3 IBConsole's Interactive SQL window lets you try in advance the queries you plan to include in your Delphi programs.

This has been a minimal description of IBConsole, which is a powerful tool (and the only one Borland includes with the server other than command-line tools). IBConsole is not the most complete tool in its category, though. Quite a few third-party InterBase management applications are more powerful, although they are not all stable or user-friendly. Some InterBase tools are shareware programs, and others are free. Two examples out of many are InterBase Workbench (`www.upscene.com`) and IB_WISQL (done with and part of InterBase Objects, `www.ibobjects.com`).

InterBase Server-Side Programming

At the beginning of this chapter, I underlined the fact that one of the objectives of client/server programming—and one of its problems—is the division of the workload between the computers involved. When you activate SQL statements from the client, the burden falls on the server to do most of the work. However, you should try to use `select` statements that return a large result set, to avoid jamming the network.

In addition to accepting DDL (Data Definition Language) and DML (Data Manipulation Language), most RDBMS servers allow you to create routines directly on the server using the standard SQL commands plus their own server-specific extensions (which generally are not portable). These routines typically come in two forms: stored procedures and triggers.

STORED PROCEDURES

Stored procedures are like the global functions of a Delphi unit and must be explicitly called by the client side. Stored procedures are generally used to define routines for data maintenance, to group sequences of operations you need in different circumstances, or to hold complex `select` statements.

Like Delphi procedures, stored procedures can have one or more typed parameters. Unlike Delphi procedures, they can have more than one return value. As an alternative to returning a value, a stored procedure can also return a result set—the result of an internal `select` statement or a custom fabricated one.

The following is a stored procedure written for InterBase; it receives a date as input and computes the highest salary among the employees hired on that date:

```
create procedure MaxSalOfTheDay (ofday date)
returns (maxsal decimal(8,2)) as
begin
  select max(salary)
  from employee
  where hiredate = :ofday
  into :maxsal;
end
```

Notice the use of the `into` clause, which tells the server to store the result of the `select` statement in the `maxsal` return value. To modify or delete a stored procedure, you can later use the `alter procedure` and `drop procedure` commands.

Looking at this stored procedure, you might wonder what its advantage is compared to the execution of a similar query activated from the client. The difference between the two approaches is not in the result you obtain but in its speed. A stored procedure is compiled on the server in an intermediate and faster notation when it is created, and the server determines at that time the strategy it will use to access the data. By contrast, a query is compiled every time the request is sent to the server. For this reason, a stored procedure can replace a very complex query, provided it doesn't change too often.

From Delphi, you can activate a stored procedure with the following SQL code:

```
select *
from MaxSalOfTheDay ('01/01/2003')
```

TRIGGERS (AND GENERATORS)

Triggers behave more or less like Delphi events and are automatically activated when a given event occurs. Triggers can have specific code or call stored procedures; in both cases, the execution is done completely on the server. Triggers are used to keep data consistent, checking new data in more complex ways than a check constraint allows, and to automate the side effects of some input operations (such as creating a log of previous salary changes when the current salary is modified).

Triggers can be fired by the three basic data update operations: `insert`, `update`, and `delete`. When you create a trigger, you indicate whether it should fire before or after one of these three actions.

As an example of a trigger, you can use a generator to create a unique index in a table. Many tables use a unique index as a primary key. InterBase doesn't have an AutoInc field. Because multiple clients cannot generate unique identifiers, you can rely on the server to do this. Almost all SQL servers offer a counter you can call to ask for a new ID, which you should later use for the table. InterBase calls these automatic counters *generators,* and Oracle calls them *sequences.* Here is the sample InterBase code:

```
create generator cust_no_gen;
...
gen_id (cust_no_gen, 1);
```

The `gen_id` function extracts the new unique value of the generator passed as the first parameter; the second parameter indicates how much to increase (in this case, by one).

At this point you can add a trigger to a table (an automatic handler for one of the table's events). A trigger is similar to the event handler of the Table component, but you write it in SQL and execute it on the server, not on the client. Here is an example:

```
create trigger set_cust_no for customers
before insert position 0 as
begin
  new.cust_no = gen_id (cust_no_gen, 1);
end
```

This trigger is defined for the customers table and is activated each time a new record is inserted. The `new` symbol indicates the new record you are inserting. The `position` option indicates the order of execution of multiple triggers connected to the same event. (Triggers with the lowest values are executed first.)

Inside a trigger, you can write DML statements that also update other tables, but watch out for updates that end up reactivating the trigger and create endless recursion. You can later modify or disable a trigger by calling the `alter trigger` or `drop trigger` statement.

Triggers fire automatically for specified events. If you have to make many changes in the database using batch operations, the presence of a trigger can slow the process. If the input data has already been checked for consistency, you can temporarily deactivate the trigger. These batch operations are often coded in stored procedures, but stored procedures generally cannot issue DDL statements like those required for deactivating and reactivating the trigger. In this situation, you can define a view based on a `select * from table` command, thus creating an alias for the table. Then you can let the stored procedure do the batch processing on the table and apply the trigger to the view (which should also be used by the client program).

The dbExpress Library

Nowadays, the mainstream access to a SQL server database in Delphi is provided by the dbExpress library. As mentioned in Chapter 13, "Delphi's Database Architecture," this is not the only possibility but is certainly the mainstream approach. The dbExpress library, first introduced in Kylix and Delphi 6, allows you to access different servers (InterBase, Oracle, DB2, MySql, Informix, and now Microsoft SQL Server). I provided a general overview of dbExpress compared with other solutions in Chapter 13, so here I'll skip the introductory material and focus on technical elements.

NOTE *The inclusion of a driver for Microsoft SQL Server is the most important update to dbExpress provided by Delphi 7. It is not implemented by interfacing the vendor library natively, like other dbExpress drivers, but by interfacing Microsoft's OLEDB provider for SQL Server. (I'll talk more about OLEDB providers in Chapter 15.)*

Working with Unidirectional Cursors

The motto of dbExpress could be "fetch but don't cache." The key difference between this library and BDE or ADO is that dbExpress can only execute SQL queries and fetch the results with a *unidirectional cursor*. As you've just seen, in unidirectional database access, you can move from one record to the next, but you cannot get back to a previous record of the dataset (unless by reopening the query and fetching all the records again minus one, an incredibly slow operation that dbExpress blocks). This is because the library doesn't store the data it has retrieved in a local cache, but only passes it from the database server to the calling application.

Using a unidirectional cursor might sound like a limitation, and it is—in addition to having problems with navigation, you cannot connect a database grid to a dataset. However, a unidirectional dataset is good for the following:

- You can use a unidirectional dataset for reporting purposes. In a printed report, but also an HTML page or an XML transformation, you move from record to record, produce the output, and that's it—no need to return to past records and, in general, no user interaction with the data. Unidirectional datasets are probably the best option for web and multitier architectures.
- You can use a unidirectional dataset to feed a local cache, such as the one provided by a ClientDataSet component. At this point, you can connect visual components to the in-memory dataset and operate on it with all the standard techniques, including the use of visual grids. You can freely navigate and edit the data in the in-memory cache, but also control it far better than with the BDE or ADO.

It's important to notice that in these circumstances, avoiding the caching of the database engine saves time and memory. The library doesn't have to use extra memory for the cache and doesn't need to waste time storing data (and duplicating information). Over the last couple of years, many programmers moved from BDE-based cached updates to the ClientDataSet component, which provides more flexibility in managing the content of the data and updating information they keep in memory. However, using a ClientDataSet on top of the BDE (or ADO) exposes you to the risk of having two separate caches, which wastes a lot of memory.

Another advantage of using the ClientDataSet component is that its cache supports editing operations, and the updates stored in this cache can be applied to the original database server by the DataSetProvider component. This component can generate the proper SQL `update` statements, and can do so in a more flexible way than the BDE (although ADO is powerful in this respect). In general, the provider can also use a dataset for the updates, but this isn't directly possible with the dbExpress dataset components.

Platforms and Databases

A key element of the dbExpress library is its availability for both Windows and Linux, in contrast to the other database engines available for Delphi (BDE and ADO), which are available only for Windows. However, some of the database-specific components, such as InterBase Express, are also available on multiple platforms.

When you use dbExpress, you are provided with a common framework, which is independent of the SQL database server you are planning to use. dbExpress comes with drivers for MySQL, InterBase, Oracle, Informix, Microsoft SQL Server, and IBM DB2.

NOTE *It is possible to write custom drivers for the dbExpress architecture. This is documented in detail in the paper "dbExpress Draft Specification," published on the Borland Community website. At the time of this writing, this document is at* `http://community.borland.com/article/0,1410,22495,00.html`. *You'll probably be able to find third-party drivers. For example, there is a free driver that bridges dbExpress and ODBC. A complete list is hosted in the article at* `http://community.borland.com/article/0,1410,28371,00.html`.

Driver Versioning Troubles and Embedded Units

Technically, the dbExpress drivers are available as separate DLLs you have to deploy along with your program. This was the case with Delphi 6 and is still the case with Delphi 7. The problem is, these DLLs' names haven't changed. So, if you install a Delphi 7 compiled application on a machine that has the dbExpress drivers found in Delphi 6, the application will apparently work, open a connection to the server, and then fail when retrieving data. At that point you'll see the error "SQL Error: Error mapping failed." This is not a good hint that there is a version mismatch in the dbExpress driver!

To verify this problem, look at whether the DLL has any version information—it was missing from the Delphi 6 drivers. To make your applications more robust, you can provide a similar check within your code, accessing the version information using the related Windows APIs:

```
function GetDriverVersion (strDriverName: string): Integer;
var
  nInfoSize, nDetSize: DWord;
  pVInfo, pDetail: Pointer;
begin
  // the default, in case there is no version information
  Result := 6;

  // read version information
  nInfoSize := GetFileVersionInfoSize (pChar(strDriverName), nDetSize);
  if nInfoSize > 0 then
  begin
    GetMem (pVInfo, nInfoSize);
    try
      GetFileVersionInfo (pChar(strDriverName), 0, nInfoSize, pVInfo);
      VerQueryValue (pVInfo, '\', pDetail, nDetSize);
      Result := HiWord (TVSFixedFileInfo(pDetail^).dwFileVersionMS);
    finally
      FreeMem (pVInfo);
    end;
  end;
end;
```

This code snippet is taken from the DbxMulti example discussed later. The program uses it to raise an exception if the version is incompatible:

```
if GetDriverVersion ('dbexpint.dll') <> 7 then
  raise Exception.Create (
    'Incompatible version of the dbExpress driver "dbexpint.dll" found');
```

If you try to put the driver found in Delphi 6's `bin` folder in the application folder, you'll see the error. You'll have to modify this extra safety check to account for updated versions of the drivers or libraries, but this step should help you avoid the installation troubles dbExpress meant to solve in the first place.

You also have an alternative: You can statically link the dbExpress drivers' code into your application. To do so, include a given unit (like `dbexpint.dcu` or `dbexpora.dcu`) in your program, listing it in one of the `uses` statements.

WARNING *Along with one of these units you need to include the MidasLib unit and link the code of the MIDAS.DLL into your program. If you fail to do so, the linker of Delphi 7 will stop showing an internal error, which is rather pointless information. Notice also that the embedded dbExpress drivers don't work properly with the international character set.*

The dbExpress Components

The VCL components used to interface the dbExpress library encompass a group of dataset components plus a few ancillary ones. To differentiate these components from other database-access families, the components are prefixed with the letters *SQL*, underlining the fact that they are used for accessing RDBMS servers.

These components include a database connection component, a few dataset components (a generic one; three specific versions for tables, queries, and stored procedures; and one encapsulating a ClientDataSet component), and a monitor utility.

The SQLConnection Component

The `TSQLConnection` class inherits from the `TCustomConnection` component. It handles database connections, the same as its sibling classes (the Database, ADOConnection, and IBConnection components).

TIP *Unlike other component families, in dbExpress the connection is compulsory. In the dataset components, you cannot specify directly which database to use, but can only refer to a SQLConnection.*

The connection component uses the information available in the `drivers.ini` and `connections.ini` files, which are dbExpress's only two configuration files (these files are saved by default under `Common Files\Borland Shared\DBExpress`). The `drivers.ini` file lists the available dbExpress drivers, one for each supported database. For each driver there is a set of default connection parameters. For example, the InterBase section reads as follows:

```
[Interbase]
GetDriverFunc=getSQLDriverINTERBASE
LibraryName=dbexpint.dll
VendorLib=GDS32.DLL
BlobSize=-1
CommitRetain=False
Database=database.gdb
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
SQLDialect=1
Interbase TransIsolation=ReadCommited
User_Name=sysdba
WaitOnLocks=True
```

The parameters indicate the dbExpress driver DLL (the `LibraryName` value), the entry function to use (`GetDriverFunc`), the vendor client library, and other specific parameters that depend on the database. If you read the entire `drivers.ini` file, you'll see that the parameters are really database-specific. Some of these parameters don't make a lot of sense at the driver level (such as the database to connect to), but the list includes all the available parameters, regardless of their usage.

The `connections.ini` file provides the database-specific description. This list associates settings with a name, and you can enter multiple connection details for every database driver. The connection describes the physical database you want to connect to. As an example, this is the portion for the default `IBLocal` definition:

```
[IBLocal]
BlobSize=-1
CommitRetain=False
Database=C:\Program Files\Common Files\Borland Shared\Data\employee.gdb
DriverName=Interbase
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
SQLDialect=1
Interbase TransIsolation=ReadCommited
User_Name=sysdba
WaitOnLocks=True
```

As you can see by comparing the two listings, this is a subset of the driver's parameters. When you create a new connection, the system will copy the default parameters from the driver; you can then edit them for the specific connection—for example, providing a proper database name. Each connection relates to the driver for its key attributes, as indicated by the `DriverName` property. Notice also that the database referenced here is the result of my editing, corresponding to the settings I'll use in most examples.

It's important to remember that these initialization files are used only at design time. When you select a driver or a connection at design time, the values of these files are copied to corresponding properties of the SQLConnection component, as in this example:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  DriverName = 'Interbase'
  GetDriverFunc = 'getSQLDriverINTERBASE'
  LibraryName = 'dbexpint.dll'
  LoginPrompt = False
  Params.Strings = (
    'BlobSize=-1'
    'CommitRetain=False'
    'Database=C:\Program Files\Common Files\Borland Shared\Data\employee.gdb'
    'DriverName=Interbase'
    'Password=masterkey'
    'RoleName=RoleName'
    'ServerCharSet=ASCII'
    'SQLDialect=1'
    'Interbase TransIsolation=ReadCommited'
```

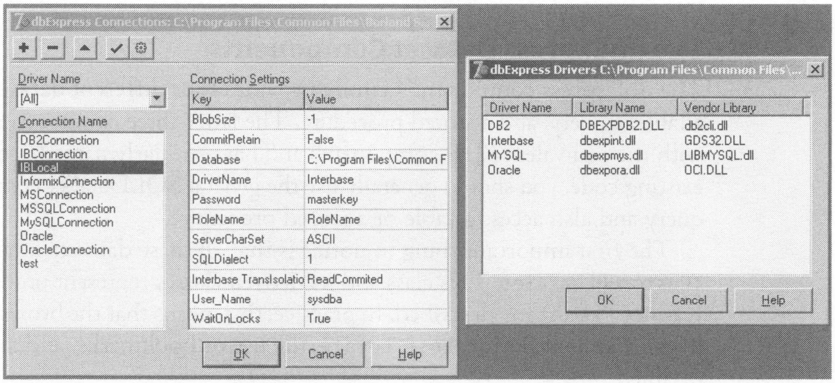
```
      'User_Name=sysdba'
      'WaitOnLocks=True')
    VendorLib = 'GDS32.DLL'
  end
```

At run time, your program will rely on the properties to have all the required information, so you don't need to deploy the two configuration files along with your programs. In theory, the files will be required if you want to change the `DriverName` or `ConnectionName` properties at run time. However, if you want to connect your program to a new database, you can set the relevant properties directly.

When you add a new SQLConnection component to an application, you can proceed in different ways. You can set up a driver using the list of values available for the `DriverName` property and then select a predefined connection by selecting one of the values available in the `ConnectionName` property. This second list is filtered according to the driver you've already selected. As an alternative, you can begin by selecting the `ConnectionName` property directly; in this case it includes the entire list.

Instead of hooking up an existing connection, you can define a new one (or see the details of the existing connections) by double-clicking the SQLConnection component and launching the dbExpress Connection Editor (see Figure 14.4). This editor lists on the left all the predefined connections (for a specific driver or all of them) and allows you to edit the connection properties using the grid on the right. You can use the toolbar buttons to add, delete, rename, and test connections, and to open the read-only dbExpress Drivers Settings window (also shown in Figure 14.4).

FIGURE 14.4
The dbExpress Connection Editor with the dbExpress Drivers Settings dialog box


dbExpress Connections: C:\Program Files\Common Files\Borland S...
Driver Name
[All]
Connection Name
DB2Connection
IBConnection
IBLocal
InformixConnection
MSConnection
MSSQLConnection
MySQLConnection
Oracle
OracleConnection
test
Connection Settings
OK
Cancel
Help
dbExpress Drivers C:\Program Files\Common Files\...
OK
Cancel
Help

Key	Value
BlobSize	-1
CommitRetain	False
Database	C:\Program Files\Common F
DriverName	Interbase
Password	masterkey
RoleName	RoleName
ServerCharSet	ASCII
SQLDialect	1
Interbase TransIsolatio	ReadCommited
User_Name	sysdba
WaitOnLocks	True

Driver Name	Library Name	Vendor Library
DB2	DBEXPDB2.DLL	db2cli.dll
Interbase	dbexpint.dll	GDS32.DLL
MYSQL	dbexpmys.dll	LIBMYSQL.dll
Oracle	dbexpora.dll	OCI.DLL

In addition to letting you edit the predefined connection settings, the dbExpress Connection Editor allows you to select a connection for the SQLConnection component by clicking the OK button. Note that if you change any settings, the data is immediately written to the configuration files—clicking the Cancel button doesn't undo your editing!

To define access to a database, editing the connection properties is certainly the suggested approach. This way, when you need to access the same database from another application or another connection within the same application, all you need to do is select the connection. However, because this operation copies the connection data, updating the connection doesn't automatically refresh the values within other SQLConnection components referring to the same named connection: You must reselect the connection to which these other components refer.

What really matters for the SQLConnection component is the value of its properties. Driver and vendor libraries are listed in properties you can freely change at design time (although you'll rarely want to do this), whereas the database and other database-specific connection settings are specified in the `Params` properties. This is a string list including information such as the database name, the username and password, and so on. In practice, you could set up a SQLConnection component by setting up the driver and then assigning the database name directly in the `Params` property, forgetting about the predefined connection. I'm not suggesting this as the best option, but it is certainly a possibility; the predefined connections are handy, but when the data changes, you still have to manually refresh every SQLConnection component.

To be complete, I have to mention that there is an alternative. You can set the `LoadParamsOnConnect` property to indicate that you want to refresh the component parameters from the initialization files every time you open the connection. In this case, a change in the predefined connections will be reloaded when you open the connection, at either design time or run time. At design time, this technique is handy (it has the same effect as reselecting the connection); but using it at run time means you'll also have to deploy the `connections.ini` file, which can be a good idea or inconvenient, depending on your deployment environment.

The only property of the SQLConnection component that is not related to the driver and database settings is `LoginPrompt`. Setting it to False allows you to provide a password among the component settings and skip the login request dialog box, both at design time and at run time. Although this is handy for development, it can reduce the security of your system. Of course, you should also use this option for unattended connections, such as on a web server.

The dbExpress Dataset Components

The dbExpress component's family provides four different dataset components: a generic dataset, a table, a query, and a stored procedure. The latter three components are provided for compatibility with the equivalent BDE components and have similarly named properties. If you don't have to port existing code, you should generally use the general SQLDataSet component, which lets you execute a query and also access a table or a stored procedure.

The first important thing to notice is that all these datasets inherit from a new special base class, `TCustomSQLDataSet`. This class and its derived classes represent unidirectional datasets, with the key features I've already described. In practice, this means that the browse operations are limited to calling `First` and `Next`; `Prior`, `Last`, `Locate`, the use of bookmarks, and all other navigational features are disabled.

NOTE *Technically, some of the moving operations call the* `CheckBiDirectional` *internal function and eventually raise an exception.* `CheckBiDirectional` *refers to the public* `IsUnidirectional` *property of the* `TDataSet` *class, which you can eventually use in your own code to disable operations that are illegal on unidirectional datasets.*

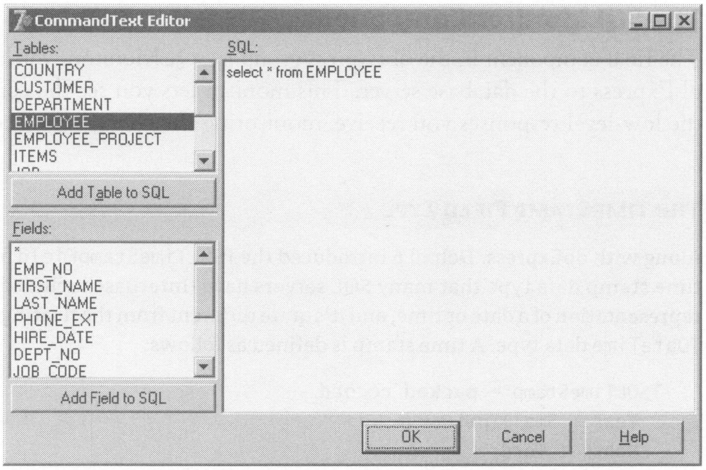
In addition to having limited navigational capabilities, these datasets have no editing support, so a lot of methods and events common to other datasets are not available. For example, there is no `AfterEdit` or `BeforePost` event.

As I mentioned earlier, of the four dataset components for dbExpress, the fundamental one is `TSQLDataSet`, which can be used both to retrieve a dataset and to execute a command. The two alternatives are activated by calling the `Open` method (or setting the `Active` property to True) and by calling the `ExecSQL` method.

The SQLDataSet component can retrieve an entire table, or it can use a SQL query or a stored procedure to read a dataset or issue a command. The `CommandType` property determines one of the three access modes. The possible values are ctQuery, ctStoredProc, and ctTable, which determine the value of the `CommandText` property (and also the behavior of the related property editor in the Object Inspector). For a table or stored procedure, the `CommandText` property indicates the name of the related database element, and the editor provides a drop-down list containing the possible values. For a query, the `CommandText` property stores the text of the SQL command, and the editor provides a little help in building the SQL query (in case it is a SELECT statement). You can see the editor in Figure 14.5.

FIGURE 14.5

The CommandText Editor used by the SQLDataSet component for queries



```
select * from EMPLOYEE
```

When you use a table, the component will generate a SQL query for you, because dbExpress targets only SQL databases. The generated query will include all the fields of the table, and if you specify the `SortFieldNames` property, it will include a `sort by` directive.

The three *specific* dataset components offer similar behavior, but you specify the SQL query in the `SQL` string list property, the stored procedure in the `StoredProcName` property, and the table name in the `TableName` property (as in the three corresponding BDE components).

THE DELPHI 7 SIMPLEDATASET COMPONENT

The SimpleDataSet component is new in Delphi 7. It is a combination of four existing components: SQLConnection, SQLDataSet, DataSetProvider, and ClientDataSet. The component is meant to be a helper—you need only one component instead of four (which must also be connected). The component is basically a client dataset with two compound components (the two dbExpress ones), plus a hidden provider. (The fact that the provider is hidden is odd, because it is created as a compound component.)

The component allows you to modify the properties and events of the compound components (besides the provider) and replace the internal connection with an external one, so that multiple datasets share the same database connection. In addition to this issue, the component has other limitations, including difficulty manipulating the dataset fields of the data access dataset (which is important for setting key fields and can affect the way updates are generated) and unavailability of some provider events. So, other than for simple applications, I don't recommend using the SimpleDataSet component.

NOTE *Delphi 6 shipped with an even simpler and more limited component called SQLClientDataSet. Similar components were available for the BDE and IBX data access technologies. Now Borland has indicated that all these components are obsolete. However, the directory* `Demos\Db\SQLClientDataset` *contains a copy of the original component, and you can install it in Delphi 7 for compatibility purposes. But just as the SimpleDataSet component is somewhat limited, I found the SQLClientDataSet component totally unusable.*

The SQLMonitor Component

The final component in the dbExpress group is SQLMonitor, which is used to log requests sent from dbExpress to the database server. This monitor lets you see the commands sent to the database and the low-level responses you receive, monitoring the client/server traffic at a low level.

THE TIMESTAMP FIELD TYPE

Along with dbExpress, Delphi 6 introduced the `TSQLTimeStampField` field type, which is mapped to the time stamp data type that many SQL servers have (InterBase included). A *time stamp* is a record-based representation of a date or time, and it's quite different from the floating-point representation used by the `TDateTime` data type. A time stamp is defined as follows:

```
TSQLTimeStamp = packed record
  Year : SmallInt;
  Month : Word;
  Day : Word;
  Hour : Word;
  Minute : Word;
  Second : Word;
  Fractions : LongWord;
end;
```

A time stamp field can automatically convert standard date and time values using the `AsDateTime` property (as opposed to the native `AsSQLTimeStamp` property). You can also do custom conversions and further manipulation of time stamps using the routines provided by the SqlTimSt unit, including functions like `DateTimeToSQLTimeStamp`, `SQLTimeStampToStr`, and `VarSQLTimeStampCreate`.

A Few dbExpress Demos

Let's look at a demonstration that highlights the key features of these components and shows how to use the ClientDataSet to provide caching and editing support for the unidirectional datasets. Later, I'll show you an example of native use of the unidirectional query, with no caching and editing support required.

The standard visual application based on dbExpress uses this series of components:

- The SQLConnection component provides the connection with the database and the proper dbExpress driver.
- The SQLDataSet component, which is hooked to the connection (via the `SQLConnection` property), indicates which SQL query to execute or table to open (using the `CommandType` and `CommandText` properties discussed earlier).
- The DataSetProvider component, connected with the dataset, extracts the data from the SQLDataSet and can generate the proper SQL update statements.
- The ClientDataSet component reads from the data provider and stores all the data (if its `PacketRecords` property is set to –1) in memory. You'll need to call its `ApplyUpdates` method to send the updates back to the database server (through the provider).
- The DataSource component allows you to surface the data from the ClientDataSet to the visual data-aware controls.

As I mentioned earlier, the picture can be simplified by using the SimpleDataSet component, which replaces the two datasets and the provider (and possibly even the connection). The SimpleDataSet component combines most of the properties of the components it replaces.

Using a Single Component or Many Components

For this first example, drop a SimpleDataSet component on a form and set the connection name of its Connection subcomponent. Set the `CommandType` and `CommandText` properties to specify which data to fetch, and set the `PacketRecords` property to indicate how many records to retrieve in each block.

These are the key properties of the component in the DbxSingle example:

```
object SimpleDataSet1: TSimpleDataSet
  Connection.ConnectionName = 'IBLocal'
  Connection.LoginPrompt = False
  DataSet.CommandText = 'EMPLOYEE'
  DataSet.CommandType = ctTable
end
```

As an alternative, the DbxMulti example uses the entire sequence of components:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText = 'select * from EMPLOYEE'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
```

```
object ClientDataSet1: TClientDataSet
  ProviderName = 'DataSetProvider1'
end
object DataSource1: TDataSource
  DataSet = ClientDataSet1
end
```

Both examples include some visual controls: a grid and a toolbar based on the action manager architecture.

APPLYING UPDATES

In every example based on a local cache, like the one provided by the ClientDataSet and SimpleDataSet components, it's important to write the local changes back to the database server. This is typically accomplished by calling the `ApplyUpdates` method. You can either keep the changes in the local cache for a while and then apply multiple updates at once, or you can post each change right away. In these two examples, I've gone for the latter approach, attaching the following event handler to the `AfterPost` (fired after an edit or an insert operation) and `AfterDelete` events of the ClientDataSet components:

```
procedure TForm1.DoUpdate(DataSet: TDataSet);
begin
  // immediately apply local changes to the database
  SQLClientDataSet1.ApplyUpdates(0);
end;
```

If you want to apply all the updates in a single batch, you can do so either when the form is closed or when the program ends, or you can let a user perform the update operation by selecting a specific command, possibly using the corresponding predefined action provided by Delphi 7. We'll explore this approach when discussing the update caching support of the ClientDataSet component in more detail later in this chapter.

MONITORING THE CONNECTION

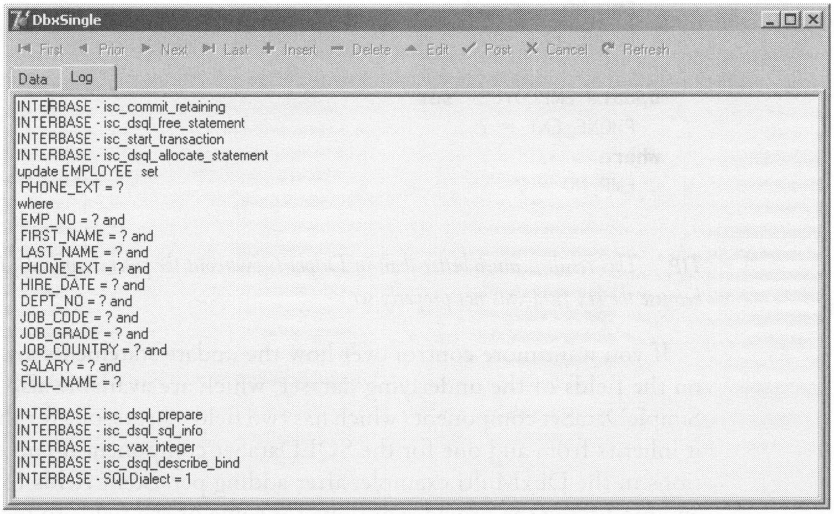
Another feature I've added to the DbxSingle and DbxMulti examples is the monitoring capability offered by the SQLMonitor component. In the example, the component is activated as the program starts. In the DbxSingle example, because the SimpleDataSet embeds the connection, the monitor cannot be hooked to it at design time, but only when the program starts:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  SQLMonitor1.SQLConnection := SimpleDataSet1.Connection;
  SQLMonitor1.Active := True;
  SimpleDataSet1.Active := True;
end;
```

Every time a tracing string is available, the component fires the `OnTrace` event to let you choose whether to include the string in the log. If the `LogTrace` parameter of this event is True (the default value), the component logs the message in the `TraceList` string list and fires the `OnLogTrace` event to indicate that a new string has been added to the log.

The component can also automatically store the log into the file indicated by its `FileName` property, but I haven't used this feature in the example. All I've done is handle the `OnTrace` event, copying the entire log in the memo with the following code (producing the output shown in Figure 14.6):

```
procedure TForm1.SQLMonitor1Trace(Sender: TObject;
  CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
begin
  Memo1.Lines := SQLMonitor1.TraceList;
end;
```



```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_start_transaction
INTERBASE - isc_dsql_allocate_statement
update EMPLOYEE  set
 PHONE_EXT = ?
where
 EMP_NO = ? and
 FIRST_NAME = ? and
 LAST_NAME = ? and
 PHONE_EXT = ? and
 HIRE_DATE = ? and
 DEPT_NO = ? and
 JOB_CODE = ? and
 JOB_GRADE = ? and
 JOB_COUNTRY = ? and
 SALARY = ? and
 FULL_NAME = ?

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
```

FIGURE 14.6
A sample log obtained by the SQLMonitor in the DbxSingle example

CONTROLLING THE SQL UPDATE CODE

If you run the DbxSingle program and change, for example, an employee's telephone number, the monitor will log this update operation:

```
update EMPLOYEE  set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  FIRST_NAME = ? and
  LAST_NAME = ? and
  PHONE_EXT = ? and
  HIRE_DATE = ? and
  DEPT_NO = ? and
  JOB_CODE = ? and
  JOB_GRADE = ? and
  JOB_COUNTRY = ? and
  SALARY = ? and
  FULL_NAME = ?
```

By setting the SimpleDataSet's properties there is no way to change how the update code is generated (which happens to be worse than with the SQLClientDataSet component, which had the `UpdateMode` you could use to tweak the update statements).

In the DbxMulti example, you can use the `UpdateMode` property of the DataSetProvider component, setting the value to upWhereChanged or upWhereKeyOnly. In this case you'll get the following two statements, respectively:

```
update EMPLOYEE  set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  PHONE_EXT = ?

update EMPLOYEE  set
  PHONE_EXT = ?
where
  EMP_NO = ?
```

TIP *This result is much better than in Delphi 6 (without the patches applied), in which this operation caused an error because the key field was not properly set.*

If you want more control over how the update statements are generated, you need to operate on the fields of the underlying dataset, which are available also when you use the all-in-one SimpleDataSet component (which has two field editors, one for the base ClientDataset component it inherits from and one for the SQLDataSet component it embeds). I have made similar corrections in the DbxMulti example, after adding persistent fields for the SQLDataSet component and modifying the provider options for some of the fields to include them in the key or exclude them from updates.

NOTE *We'll discuss this type of problem again when we examine the details of the ClientDataSet component, the provider, the resolver, and other technical details later in this chapter and in Chapter 16.*

Accessing Database Metadata with *SetSchemaInfo*

All RDBMS systems use special-purpose tables (generally called *system tables*) for storing metadata, such as the list of the tables, their fields, indexes, and constraints, and any other system information. Just as dbExpress provides a unified API for working with different SQL servers, it also provides a common way to access metadata. The SQLDataSet component has a `SetSchemaInfo` method that fills the dataset with system information. This `SetSchemaInfo` method has three parameters:

SchemaType Indicates the type of information requested. Values include stTables, stSysTables, stProcedures, stColumns, and stProcedureParams.

SchemaObject Indicates the object you are referring to, such as the name of the table whose columns you are requesting.

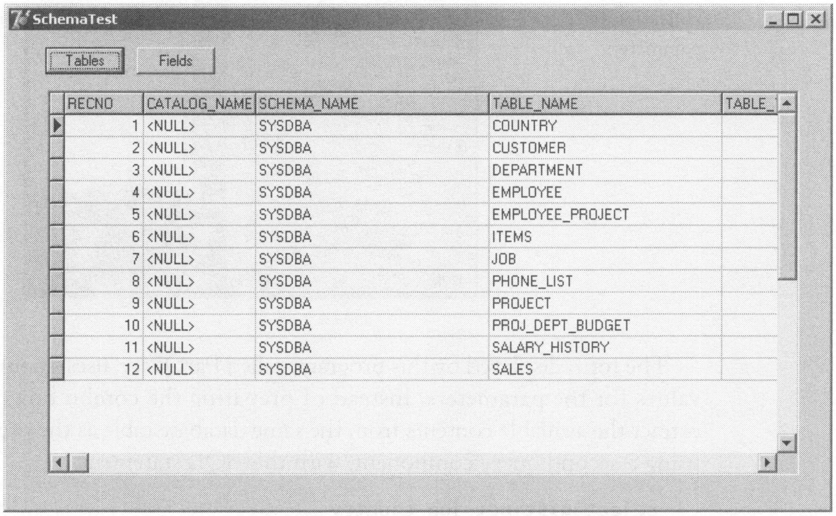
SchemaPattern A filter that lets you limit your request to tables, columns, or procedures starting with the given letters. This is handy if you use prefixes to identify groups of elements.

For example, in the SchemaTest program, a Tables button reads into the dataset all of the connected database's tables:

```
ClientDataSet1.Close;
SQLDataSet1.SetSchemaInfo (stTables, '', '');
ClientDataSet1.Open;
```

The program uses the usual group of dataset provider, client dataset, and data source component to display the resulting data in a grid, as you can see in Figure 14.7. After you're retrieved the tables, you can select a row in the grid and click the Fields button to see a list of the fields of this table:

```
SQLDataSet1.SetSchemaInfo (stColumns, ClientDataSet1['Table_Name'], '');
ClientDataSet1.Close;
ClientDataSet1.Open;
```



RECNO	CATALOG_NAME	SCHEMA_NAME	TABLE_NAME	TABLE_
1	<NULL>	SYSDBA	COUNTRY	
2	<NULL>	SYSDBA	CUSTOMER	
3	<NULL>	SYSDBA	DEPARTMENT	
4	<NULL>	SYSDBA	EMPLOYEE	
5	<NULL>	SYSDBA	EMPLOYEE_PROJECT	
6	<NULL>	SYSDBA	ITEMS	
7	<NULL>	SYSDBA	JOB	
8	<NULL>	SYSDBA	PHONE_LIST	
9	<NULL>	SYSDBA	PROJECT	
10	<NULL>	SYSDBA	PROJ_DEPT_BUDGET	
11	<NULL>	SYSDBA	SALARY_HISTORY	
12	<NULL>	SYSDBA	SALES	

FIGURE 14.7
The SchemaTest example allows you to see a database's tables and the columns of a given table.

In addition to letting you access database metadata, dbExpress provides a way to access its own configuration information, including the installed drivers and the configured connections. The unit DbConnAdmin defines a `TConnectionAdmin` class for this purpose, but the aim of this support is limited to dbExpress add-on utilities for developers (end users aren't commonly allowed to access multiple databases in a totally dynamic way).

TIP *The DbxExplorer demo included in Delphi shows how to access both dbExpress administration files and schema information. Also check the help file under "The structure of metadata datasets" within the section "Developing database applications."*

A Parametric Query

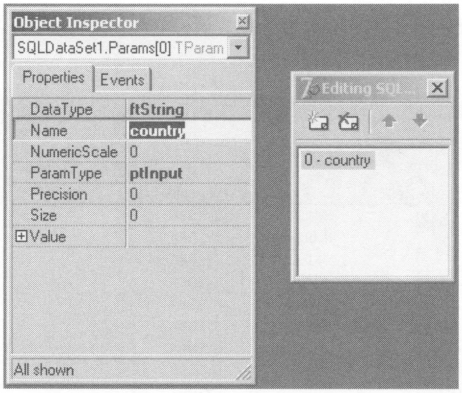
When you need slightly different versions of the same SQL query, instead of modifying the text of the query itself each time, you can write a query with a parameter and change the value of the parameter. For example, if you decide to have a user choose the employees in a given country (using the *employee* table), you can write the following parametric query:

```
select *
from employee
where job_country = :country
```

In this SQL clause, `:country` is a parameter. You can set its data type and startup value using the editor of the SQLDataSet component's `Params` property collection. When the `Params` collection editor is open, as shown in Figure 14.8, you see a list of the parameters defined in the SQL statement; you can set the data type and the initial value of these parameters in the Object Inspector.

FIGURE 14.8

Editing a query component's collection of parameters



SQLDataSet1.Params[0] TParam

DataType ftString
Name country
NumericScale 0
ParamType ptInput
Precision 0
Size 0
Value

0 - country

The form displayed by this program, called ParQuery, uses a combo box to provide all the available values for the parameters. Instead of preparing the combo box items at design time, you can extract the available contents from the same database table as the program starts. This is accomplished using a second query component, with this SQL statement:

```
select distinct job_country
from employee
```

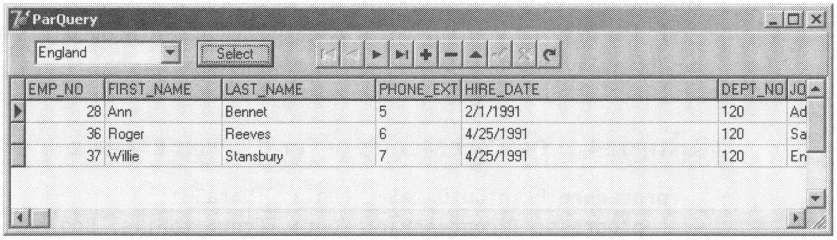
After activating this query, the program scans its result set, extracting all the values and adding them to the list box:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
  SqlDataSet2.Open;
  while not SqlDataSet2.EOF do
  begin
    ComboBox1.Items.Add (SqlDataSet2.Fields [0].AsString);
    SqlDataSet2.Next;
  end;
  ComboBox1.Text := CombBox1.Items[0];
end;
```

The user can select a different item in the combo box and then click the Select button (`Button1`) to change the parameter and activate (or re-activate) the query:

```
procedure TQueryForm.Button1Click(Sender: TObject);
begin
  SqlDataSet1.Close;
  ClientDataSet1.Close;
  Query1.Params[0].Value := ListBox1.Items [Listbox1.ItemIndex];
  SqlDataSet1.Open;
  ClientDataSet1.Open;
end;
```

This code displays the employees from the selected country in the DBGrid, as you can see in Figure 14.9. As an alternative to using the elements of the `Params` array by position, you should consider using the `ParamByName` method, to avoid any problem in case the query gets modified over time and the parameters end up in a different order.



EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE	DEPT_NO	JO
28	Ann	Bennet	5	2/1/1991	120	Ad
36	Roger	Reeves	6	4/25/1991	120	Sa
37	Willie	Stansbury	7	4/25/1991	120	En

FIGURE 14.9
The ParQuery example at run time

By using parametric queries, you can usually reduce the amount of data moved over the wire from the server to the client and still use a DBGrid and the standard user interface common in local database applications.

TIP Parametric queries are generally also used to obtain master-detail architectures with SQL queries—at least, this is what Delphi tends to do. The `DataSource` *property of the SQLDataSet component, automatically replaces parameter values with the fields of the master dataset having the same name as the parameter.*

When One-Way Is Enough: Printing Data

You have seen that one of the key elements of the dbExpress library is that it returns unidirectional datasets. In addition, you can use the ClientDataSet component (in one of its incarnations) to store the records in a local cache. Now, let's discuss an example in which a unidirectional dataset is all you need.

Such a situation is common in *reporting*—that is, producing information for each record in sequence without needing any further access to the data. This broad category includes producing printed reports (via a set of reporting components or using the printer directly), sending data to another application such as Microsoft Excel or Word, saving data to files (including HTML and XML formats), and more.

I don't want to delve into HTML and XML, so I'll present an example of printing—nothing fancy and nothing based on reporting components, just a way to produce a draft report on your monitor and printer. For this reason, I've used Delphi's most straightforward technique to produce a printout: assigning a file to the printer with the `AssignPrn` RTL procedure.

The example, called UniPrint, has a unidirectional SQLDataSet component hooked to an InterBase connection and based on the following SQL statement, which joins the employee table with the department table to display the name of the department where each employee works:

```
select d.DEPARTMENT, e.FULL_NAME, e.JOB_COUNTRY, e.HIRE_DATE
from EMPLOYEE e
inner join DEPARTMENT d on d.DEPT_NO = e.DEPT_NO
```

To handle printing, I've written a somewhat generic routine, requiring as parameters the data to print, a progress bar for status information, the output font, and the maximum format size of each field. The entire routine uses file-print support and formats each field in a fixed-size, left-aligned string, to produce a columnar type of report. The call to the `Format` function has a parametric format string that's built dynamically using the size of the field.

In Listing 14.1 you can see the code of the core `PrintOutDataSet` method, which uses three nested `try/finally` blocks to release all the resources properly:

LISTING 14.1: THE CORE METHOD OF THE UNIPRINT EXAMPLE

```
procedure PrintOutDataSet (data: TDataSet;
  progress: TProgressBar; Font: TFont; toFile: Boolean; maxSize: Integer = 30);
var
  PrintFile: TextFile;
  I: Integer;
  sizeStr: string;
  oldFont: TFontRecall;
begin
  // assign the output to a printer or a file
  if toFile then
  begin
    SelectDirectory ('Choose a folder', '', strDir);
    AssignFile (PrintFile,
      IncludeTrailingPathDelimiter(strDir) + 'output.txt');
  end
  else
    AssignPrn (PrintFile);
  // assign the printer to a file
  AssignPrn (PrintFile);
  Rewrite (PrintFile);

  // set the font and keep the original one
  oldFont := TFontRecall.Create (Printer.Canvas.Font);
  try
    Printer.Canvas.Font := Font;
    try
      data.Open;
      try
        // print header (field names) in bold
        Printer.Canvas.Font.Style := [fsBold];
        for I := 0 to data.FieldCount - 1 do
        begin
          sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
          Write (PrintFile, Format ('%-' + sizeStr + 's',
            [data.Fields[i].FieldName]));
        end;
        Writeln (PrintFile);

        // for each record of the dataset
        Printer.Canvas.Font.Style := [];
        while not data.EOF do
        begin
          // print out each field of the record
          for I := 0 to data.FieldCount - 1 do
          begin
            sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
            Write (PrintFile, Format ('%-' + sizeStr + 's',
              [data.Fields[i].AsString]));
          end;
          Writeln (PrintFile);
          // advance ProgressBar
          progress.Position := progress.Position + 1;
          data.Next;
        end;
      finally
        // close the dataset
        data.Close;
      end;
    finally
      // reassign the original printer font
      oldFont.Free;
    end;
  finally
    // close the printer/file
    CloseFile (PrintFile);
  end;
end;
```

The program invokes this routine when you click the Print All button. It executes a separate query (`select count(*) from EMPLOYEE`), which returns the number of records in the employee table. This query is necessary to set up the progress bar (the unidirectional dataset has no way of knowing how many records it will retrieve until it has reached the last one). Then it sets the output font, possibly using a fixed-width font, and calls the `PrintOutDataSet` routine:

```
procedure TNavigator.PrintAllButtonClick(Sender: TObject);
var
  Font: TFont;
begin
  // set ProgressBar range
  EmplCountData.Open;
  try
    ProgressBar1.Max := EmplCountData.Fields[0].AsInteger;
  finally
    EmplCountData.Close;
  end;

  Font := TFont.Create;
  try
    Font.Name := 'Courier New';
    Font.Size := 9;
    PrintOutDataSet (EmplData, ProgressBar1, Font, cbFile.Checked);
  finally
    Font.Free;
  end;
end;
```

The Packets and the Cache

The ClientDataSet component reads data in packets containing the number of records indicated by the `PacketRecords` property. The default value of this property is –1, which means the provider will pull all the records at once (this is reasonable only for a small dataset). Alternatively, you can set this value to zero to ask the server for only the field descriptors and no data, or you can use any positive value to specify a number.

If you retrieve only a partial dataset, then as you browse past the end of the local cache, if the `FetchOnDemand` property is set to True (the default value), the ClientDataSet component will get more records from its source. This property also controls whether BLOB fields and nested datasets of the current records are fetched automatically (these values might not be part of the data packet, depending on the dataset provider's Options value).

If you turn off this property, you'll need to fetch more records manually by calling the `GetNextPacket` method until the method returns zero. (You call `FetchBlobs` and `FetchDetails` for these other elements.)

WARNING *Notice, by the way, that before you set an index for the data, you should retrieve the entire dataset (either by going to its last record or by setting the* `PacketRecords` *property to –1). Otherwise you'll have an odd index based on partial data.*

Manipulating Updates

One of the core ideas behind the ClientDataSet component is that it is used as a local cache to collect input from a user and then send a batch of update requests to the database. The component has both a list of the changes to apply to the database server, stored in the same format used by the ClientDataSet (accessible though the `Delta` property), and a complete update log that you can manipulate with a few methods (including an Undo capability).

TIP *In Delphi 7, the ClientDataSet component's ApplyUpdates and Undo operations are also accessible through predefined actions.*

THE STATUS OF THE RECORDS

The component lets you monitor what's going on within the data packets. The `UpdateStatus` method returns one of the following indicators for the current record:

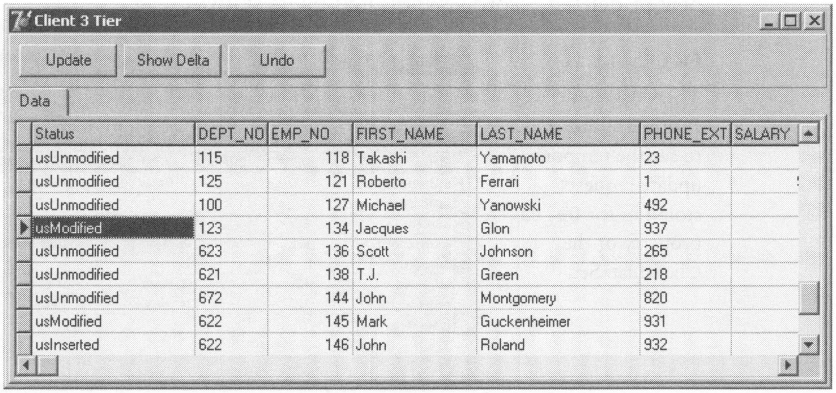
```
type TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

To easily check the status of every record in the client dataset, you can add a string-type calculated field to the dataset (I've called it `ClientDataSet1Status`) and compute its value with the following `OnCalcFields` event handler:

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  ClientDataSet1Status.AsString := GetEnumName (TypeInfo(TUpdateStatus),
    Integer (ClientDataSet1.UpdateStatus));
end;
```

This method (based on the RTTI `GetEnumName` function) converts the current value of the `TUpdateStatus` enumeration to a string, with the effect you can see in Figure 14.10.

FIGURE 14.10
The CdsDelta program displays the status of each record of a ClientDataSet.



Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	SALARY
usUnmodified	115	118	Takashi	Yamamoto	23	
usUnmodified	125	121	Roberto	Ferrari	1	
usUnmodified	100	127	Michael	Yanowski	492	
usModified	123	134	Jacques	Glon	937	
usUnmodified	623	136	Scott	Johnson	265	
usUnmodified	621	138	T.J.	Green	218	
usUnmodified	672	144	John	Montgomery	820	
usModified	622	145	Mark	Guckenheimer	931	
usInserted	622	146	John	Roland	932	

ACCESSING THE DELTA

Beyond examining the status of each record, the best way to understand which changes have occurred in a given ClientDataSet (but haven't been uploaded to the server) is to look at the *delta*—the list of changes waiting to be applied to the server. This property is defined as follows:

```
property Delta: OleVariant;
```

The format used by the `Delta` property is the same as that used for the data of a client dataset. You can add another ClientDataSet component to an application and connect it to the data in the `Delta` property of the first client dataset:

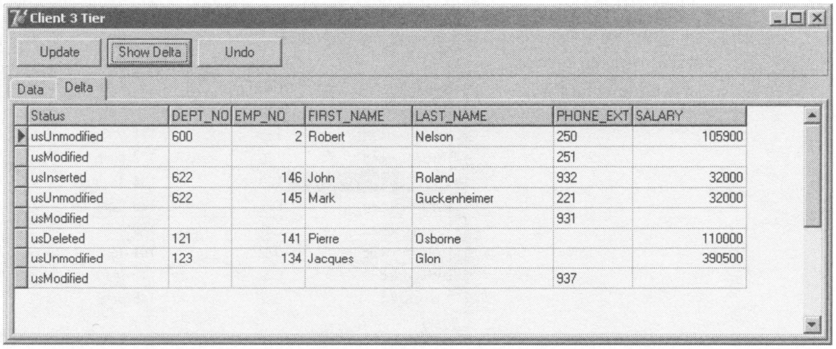
```
if ClientDataSet1.ChangeCount > 0 then
begin
  ClientDataSet2.Data := ClientDataSet1.Delta;
  ClientDataSet2.Open;
```

In the CdsDelta example, I've added a data module with the two ClientDataSet components and a source of data: a SQLDataSet mapped to InterBase's EMPLOYEE demo table. Both client datasets have the extra status calculated field, with a slightly more generic version than the code discussed earlier, because the event handler is shared between them.

TIP *To create persistent fields for the ClientDataSet hooked to the delta (at run time), I've temporarily connected it at design time to the main ClientDataSet's provider. The delta's structure is the same as the dataset it refers to. After creating the persistent fields, I removed the connection.*

The application's form has a page control with two pages, each of which has a DBGrid, one for the data and one for the delta. Code hides or shows the second tab depending on the existence of data in the change log, as returned by the `ChangeCount` method, and updates the delta when the corresponding tab is selected. The core of the code used to handle the delta is similar to the previous code snippet, and you can study the example source code on the CD to see more details.

Figure 14.11 shows the change log of the CdsDelta application. Notice that the delta dataset has two entries for each modified record (the original values and the modified fields) unless this is a new or deleted record, as indicated by its status.



Status	DEPT_NO	EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	SALARY
usUnmodified	600	2	Robert	Nelson	250	105900
usModified					251	
usInserted	622	146	John	Roland	932	32000
usUnmodified	622	145	Mark	Guckenheimer	221	32000
usModified					931	
usDeleted	121	141	Pierre	Osborne		110000
usUnmodified	123	134	Jacques	Glon		390500
usModified					937	

FIGURE 14.11
The CdsDelta example allows you to see the temporary update requests stored in the `Delta` property of the ClientDataSet.

TIP *You can filter the delta dataset (or any other ClientDataSet) depending on its update status, using the* `StatusFilter` *property. This property allows you to show new, updated, and deleted records in separate grids or in a grid filtered by selecting an option in a TabControl.*

Updating the Data

Now that you have a better understanding of what goes on during local updates, you can try to make this program work by sending the local update (stored in the delta) back to the database server. To apply all the updates from a dataset at once, pass –1 to the `ApplyUpdates` method.

If the provider (or the Resolver component inside it) has trouble applying an update, it triggers the `OnReconcileError` event. This can take place because of a concurrent update by two different people. We tend to use optimistic locking in client/server applications, so this should be regarded as a normal situation.

The `OnReconcileError` event allows you to modify the `Action` parameter (passed by reference), which determines how the server should behave:

```
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
```

This method has three parameters: the client dataset component (in case there is more than one client dataset in the current application), the exception that caused the error (with the error message), and the kind of operation that failed (ukModify, ukInsert, or ukDelete). The return value, which you'll store in the `Action` parameter, can be any one of the following:

```
type TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,
  raRefresh);
```

raSkip Specifies that the server should skip the conflicting record, leaving it in the delta (this is the default value).

raAbort Tells the server to abort the entire update operation and not try to apply the remaining changes listed in the delta.

raMerge Tells the server to merge the client data with the data on the server, applying only the modified fields of this client (and keeping the other fields modified by other clients).

raCorrect Tells the server to replace its data with the current client data, overriding all field changes already made by other clients.

raCancel Cancels the update request, removing the entry from the delta and restoring the values originally fetched from the database (thus ignoring changes made by other clients).

raRefresh Tells the server to dump the updates in the client delta and to replace them with the values currently on the server (thus keeping the changes made by other clients).

To test a collision, you can launch two copies of the client application, change the same record in both clients, and then post the updates from both. We'll do this later to generate an error, but let's first see how to handle the `OnReconcileError` event.

Handling this event is not too difficult, but only because you'll receive a little help. Because building a specific form to handle an `OnReconcileError` event is common, Delphi provides such a form in the Object Repository (available with the File ➢ New ➢ Other menu command of the Delphi IDE). Go to the Dialogs page and select the Reconcile Error Dialog item. This unit exports a function you can use directly to initialize and display the dialog box, as I've done in the CdsDelta example:

```
procedure TDmCds.cdsEmployeeReconcileError (DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

WARNING *As the source code of the Reconcile Error Dialog unit suggests, you should use the Project Options dialog to remove this form from the list of automatically created forms (if you don't, an error will occur when you compile the project). Of course, you need to do this only if you haven't set up Delphi to skip the automatic form creation.*

The `HandleReconcileError` function creates the dialog box form and shows it, as you can see in the code provided by Borland:

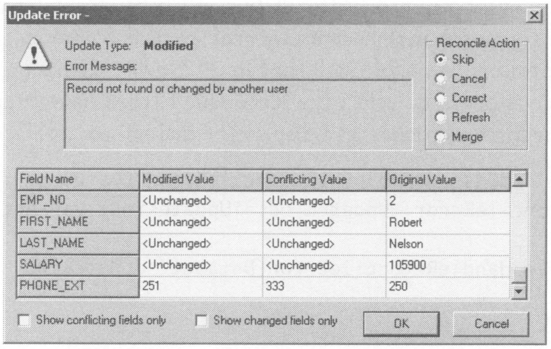
```
function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
  ReconcileError: EReconcileError): TReconcileAction;
var
  UpdateForm: TReconcileErrorForm;
begin
  UpdateForm := TReconcileErrorForm.CreateForm(DataSet, UpdateKind,
    ReconcileError);
  with UpdateForm do
  try
    if ShowModal = mrOK then
    begin
      Result := TReconcileAction(ActionGroup.Items.Objects[
        ActionGroup.ItemIndex]);
      if Result = raCorrect then
        SetFieldValues(DataSet);
    end
    else
      Result := raAbort;
  finally
    Free;
  end;
end;
```

The Reconc unit, which hosts the Reconcile Error dialog (a window titled Update Error to be more understandable by end-users of your programs), contains more than 350 lines of code, so I can't describe it in detail. However, you should be able to understand the source code by studying it carefully. Alternatively, you can use it without caring how everything works.

The dialog box will appear in case of an error, reporting the requested change that caused the conflict and allowing the user to choose one of the possible `TReconcileAction` values. You can see this form at run time in Figure 14.12.

FIGURE 14.12

The Reconcile Error dialog provided by Delphi in the Object Repository and used by the CdsDelta example



Update Type: **Modified**

Error Message:

Record not found or changed by another user

Reconcile Action
● Skip
○ Cancel
○ Correct
○ Refresh
○ Merge

Field Name	Modified Value	Conflicting Value	Original Value
EMP_NO	<Unchanged>	<Unchanged>	2
FIRST_NAME	<Unchanged>	<Unchanged>	Robert
LAST_NAME	<Unchanged>	<Unchanged>	Nelson
SALARY	<Unchanged>	<Unchanged>	105900
PHONE_EXT	251	333	250

☐ Show conflicting fields only
☐ Show changed fields only

TIP *When you call* `ApplyUpdates`*, you start a complex update sequence, which is discussed in more detail in Chapter 16 for multitier architectures. In short, the delta is sent to the provider, which fires the* `OnUpdateData` *event and then receives a* `BeforeUpdateRecord` *event for every record to update. These are two chances you have to look at the changes and force specific operations on the database server.*

Using Transactions

Whenever you are working with a SQL server, you should use *transactions* to make your applications more robust. You can think of a transaction as a series of operations that are considered a single, "atomic" whole that cannot be split.

An example may help to clarify the concept. Suppose you have to raise the salary of each employee of a company by a fixed rate, as you did in the Total example in Chapter 13. A typical program would execute a series of SQL statements on the server, one for each record to update. If an error occurred during the operation, you might want to undo the previous changes. If you consider the operation "raise the salary of each employee" as a single transaction, it should either be completely performed or completely ignored. Or, consider the analogy with financial transactions—if an error causes only part of the operation to be performed, you might end up with a missed credit or with some extra money.

Working with database operations as transactions serves a useful purpose. You can start a transaction and do several operations that should all be considered parts of a single larger operation; then, at the end, you can either commit the changes or *roll back* the transaction, discarding all the operations done up to that moment. Typically, you might want to roll back a transaction if an error occurred during its operations.

There is another important element to underline: Transactions also serve a purpose when reading data. Until data is committed by a transaction, other connections and/or transactions should not see it. Once the data is committed from a transaction, others should see the change when reading the data—that is, unless you need to open a transaction and read the same data over and over for data analysis or complex reporting operations. Different SQL servers allow you to read data in transaction according to some or all of these alternatives, as you'll see when we discuss transaction isolation levels.

Handling transactions in Delphi is simple. By default, each edit/post operation is considered a single *implicit* transaction, but you can alter this behavior by handling the operations explicitly. Use the following three methods of the dbExpress SQLConnection component (other database connection components have similar methods):

StartTransaction	Marks the beginning of a transaction
Commit	Confirms all the updates to the database done during the transaction
Rollback	Returns the database to its state prior to starting the transaction

You can also use the `InTransaction` property to check whether a transaction is active. You'll often use a `try` block to roll back a transaction when an exception is raised, or you can commit the transaction as the last operation of the `try` block, which is executed only when there is no error. The code might look like this:

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
  try
    // -- operations within the transaction go here --
    SQLConnection1.Commit(TD);
  except
     SQLConnection1.Rollback(TD);
  end;
```

Each transaction-related method has a parameter describing the transaction it is working with. The parameter uses the record type `TTransactionDesc` and accounts for a transaction isolation level and a transaction ID. The transaction isolation level is an indication of how the transaction should behave when other transactions make changes to the data. The three predefined values are as follows:

tiDirtyRead Makes the transaction's updates immediately visible to other transactions, even before they are committed. This is the only possibility in a few databases and corresponds to the behavior of databases with no transaction support.

tiReadCommitted Makes available to other transactions only the updates already committed by this transaction. This setting is recommended for most databases, to preserve efficiency.

tiRepeatableRead Hides changes made by every transaction started after the current one, even if the changes have been committed. Subsequent repeat calls within a transaction will always produce the same result, as if the database took a snapshot of the data when the current transaction started. Only InterBase and few other database servers work efficiently with this model.

TIP *As a general suggestion, for performance reasons transactions should involve a minimal number of updates (only those strictly related and part of a single atomic operation) and should be kept short in time. You should avoid transactions that wait for user input to complete them, because the user might be temporarily gone, and the transaction might remain active for a long time. Caching changes locally, as the ClientDataSet allows, can help you make the transactions small and fast, because you can open a transaction for reading, close it, and then open a transaction for writing out the entire batch of changes.*

The other field of the `TTransactionDesc` record holds a transaction ID. It is useful only in conjunction with a database server supporting multiple concurrent transactions over the same connection, like InterBase does. You can ask the connection component whether the server supports multiple transactions or doesn't support transactions at all, using the `MultipleTransactionsSupported` and `TransactionsSupported` properties.

When the server supports multiple transactions, you must supply each transaction with a unique identifier when calling the `StartTransaction` method:

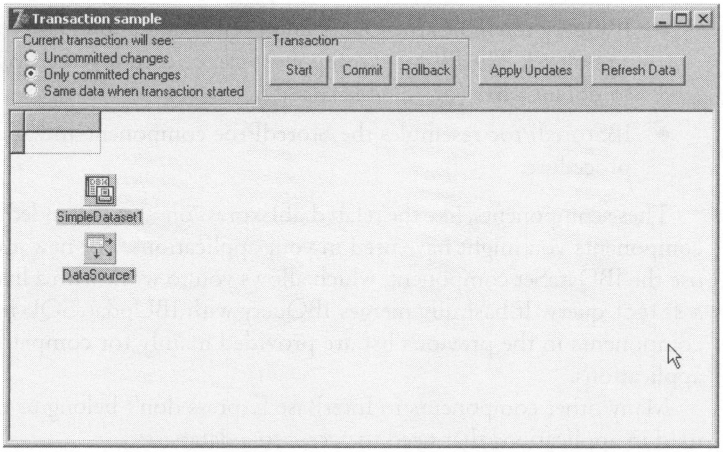
```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := GetTickCount;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
  SQLDataSet1.TransactionLevel := TD.TransactionID;
```

You can also indicate which datasets belong to which transaction by setting the `TransactionLevel` property of each dataset to a transaction ID, as shown in the last statement.

To further inspect transactions and to experiment with transaction isolation levels, you can use the TranSample application. As you can see in Figure 14.13, radio buttons let you choose the various isolation levels and buttons let you work on the transactions and apply updates or refresh data. To get a real idea of the different effects, you should run multiple copies of the program (provided you have enough licenses on your InterBase server).

FIGURE 14.13

The form of the TranSample application at design time. The radio buttons let you set different transaction isolation levels.



Current transaction will see:
○ Uncommitted changes
● Only committed changes
○ Same data when transaction started

NOTE *InterBase doesn't support the "dirty read" mode, so in the TranSample program you cannot use the last option unless you work with a different server.*

Using InterBase Express

The examples built earlier in this chapter were created with the new dbExpress database library. Using this server-independent approach allows you to switch the database server used by your application, although in practice doing so is often far from simple. If the application you are building will invariably use a given database, you can write programs that are tied directly to the API of the specific database server. This approach will make your programs intrinsically non-portable to other SQL servers.

Of course, you won't generally use these APIs directly, but rather base your development on dataset components that wrap these APIs and fit into Delphi and the architecture of its class library. An example of such a family of components is InterBase Express (IBX). Applications built using these components should work better and faster (even if only marginally), giving you more control over the specific features of the server. For example, IBX provides a set of administrative components specifically built for InterBase 6.

NOTE *I'll examine the IBX components because they are tied to InterBase (the database server discussed in this chapter) and because they are the only set of components available in the standard Delphi installation. Other similar sets of components (for InterBase, Oracle, and other database servers) are equally powerful and well regarded in the Delphi programmers' community. A good example (and an alternative to IBX) is InterBase Objects (*`www.ibobjects.com`*).*

IBX Dataset Components

The IBX components include custom dataset components and a few others. The dataset components inherit from the base `TDataSet` class, can use all the common Delphi data-aware controls, and provide a field editor and all the usual design-time features. You can choose among multiple dataset components. Three IBX datasets have a role and a set of properties similar to the table/query/storedproc components in the dbExpress family:

- IBTable resembles the Table component and allows you to access a single table or view.
- IBQuery resembles the Query component and allows you to execute a SQL query, returning a result set. The IBQuery component can be used together with the IBUpdateSQL component to obtain a live (or editable) dataset.
- IBStoredProc resembles the StoredProc component and allows you to execute a stored procedure.

These components, like the related dbExpress ones, are intended for compatibility with older BDE components you might have used in your applications. For new applications, you should generally use the IBDataSet component, which allows you to work with a live result set obtained by executing a `select` query. It basically merges IBQuery with IBUpdateSQL in a single component. The three components in the previous list are provided mainly for compatibility with existing Delphi BDE applications.

Many other components in InterBase Express don't belong to the dataset category, but are still used in applications that need to access to a database:

- IBDatabase acts like the DBX SQLConnection component and is used to set up the database connection. The BDE also uses the specific Session component to perform some global tasks done by the IBDatabase component.
- IBTransaction allows complete control over transactions. It is important in InterBase to use transactions explicitly and to isolate each transaction properly, using the Snapshot isolation level for reports and the Read Committed level for interactive forms. Each dataset explicitly refers to a given transaction, so you can have multiple concurrent transactions against the same database and choose which datasets take part in which transaction.
- IBSQL lets you execute SQL statements that don't return a dataset (for example, DDL requests, or `update` and `delete` statements) without the overhead of a dataset component.
- IBDatabaseInfo is used for querying the database structure and status.
- IBSQLMonitor is used for debugging the system, because the SQL Monitor debugger provided by Delphi is a BDE-specific tool.
- IBEvents receives events posted by the server.

This group of components provides greater control over the database server than you can achieve with dbExpress. For example, having a specific transaction component allows you to manage multiple concurrent transactions over one or multiple databases, as well as a single transaction spanning multiple databases. The IBDatabase component allows you to create databases, test the connection, and generally access system data, something the Database and Session BDE components don't fully provide.

TIP *IBX datasets let you set up the automatic behavior of a generator as a sort of auto-increment field. You do so by setting the* `GeneratorField` *property using its specific property editor. An example is discussed later in this chapter in the section "Generators and IDs."*

IBX Administrative Components

The InterBase Admin page of Delphi's Component Palette hosts InterBase administrative components. Although your aim is probably not to build a full InterBase console application, including some administrative features (such as backup handling or user monitoring) can make sense in applications meant for power users.

Most of these components have self-explanatory names: IBConfigService, IBBackupService, IBRestoreService, IBValidationService, IBStatisticalService, IBLogService, IBSecurityService, IBServerProperties, IBInstall, and IBUninstall. I won't build any advanced examples that use these components, because they are more focused toward the development of server management applications than client programs. However, I'll embed a couple of them in the IbxMon example discussed later in this chapter.

Building an IBX Example

To build an example that uses IBX, you'll need to place in a form (or data module) at least three components: an IBDatabase, an IBTransaction, and a dataset component (in this case an IBQuery). Any IBX application requires at least an instance of the first two components. You cannot set database connections in an IBX dataset, as you can do with other datasets. And, at least a transaction object is required even to read the result of a query.

Here are the key properties of these components in the IbxEmp example:

```
object IBTransaction1: TIBTransaction
  Active = False
  DefaultDatabase = IBDatabase1
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  SQL.Strings = (
    'SELECT * FROM EMPLOYEE')
end
object IBDatabase1: TIBDatabase
  DatabaseName = 'C:\Program Files\Common Files\Borland Shared\Data\employee.gdb'
  Params.Strings = (
    'user_name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  SQLDialect = 1
end
```

Now you can hook a DataSource component to `IBQuery1` and easily build a user interface for the application. I had to type in the pathname of the Borland sample database. However, not everyone has the `Program Files` folder, which depends on the local version of Windows, and the Borland sample data files could be installed elsewhere on the disk. You'll solve these problems in the next example.

WARNING *Notice that I've embedded the password in the code—a naïve approach to security. Not only can anyone run the program, but someone can extract the password by looking at the hexadecimal code of the executable file. I used this approach so I wouldn't need to keep typing in my password while testing the program, but in a real application you should require your users to do so to ensure the security of their data.*

Building a Live Query

The IbxEmp example includes a query that doesn't allow editing. To activate editing, you need to add an IBUpdateSQL component to the query, even if the query is trivial. Using an IBQuery that hosts the SQL `select` statement together with an IBUpdateSQL component that hosts the `insert`, `update`, and `delete` SQL statements is a typical approach from BDE applications. The similarities among these components make it easier to port an existing BDE application to this architecture. Here is the code for these components (edited for clarity):

```
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  SQL.Strings = (
    'SELECT Employee.EMP_NO, Department.DEPARTMENT, Employee.FIRST_NAME, '+
    '  Employee.LAST_NAME, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, '+
    '  Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY'
    'FROM EMPLOYEE Employee'
    '   INNER JOIN DEPARTMENT Department'
    '   ON  (Department.DEPT_NO = Employee.DEPT_NO)  '
    '   INNER JOIN JOB Job'
    '   ON (Job.JOB_CODE = Employee.JOB_CODE)  '
    '   AND (Job.JOB_GRADE = Employee.JOB_GRADE)  '
    '   AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)  '
    'ORDER BY Department.DEPARTMENT, Employee.LAST_NAME')
  UpdateObject = IBUpdateSQL1
end
object IBUpdateSQL1: TIBUpdateSQL
  RefreshSQL.Strings = (
    'SELECT Employee.EMP_NO, Employee.FIRST_NAME, Employee.LAST_NAME,'+
      'Department.DEPARTMENT, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO,'+
      'Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY'
    'FROM EMPLOYEE Employee'
    'INNER JOIN DEPARTMENT Department'
    'ON (Department.DEPT_NO = Employee.DEPT_NO)'
    'INNER JOIN JOB Job'
    'ON (Job.JOB_CODE = Employee.JOB_CODE)'
    'AND (Job.JOB_GRADE = Employee.JOB_GRADE)'
    'AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)'
    'WHERE Employee.EMP_NO=:EMP_NO')
  ModifySQL.Strings = (
    'update EMPLOYEE'
    'set'
    '  FIRST_NAME = :FIRST_NAME,'
    '  LAST_NAME = :LAST_NAME,'
    '  SALARY = :SALARY,'
    '  DEPT_NO = :DEPT_NO,'
    '  JOB_CODE = :JOB_CODE,'
    '  JOB_GRADE = :JOB_GRADE,'
    '  JOB_COUNTRY = :JOB_COUNTRY'
    'where'
    '  EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '(FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY)'
    'values'
    '(:FIRST_NAME,:LAST_NAME,:SALARY,:DEPT_NO,:JOB_CODE,:JOB_GRADE,:JOB_COUNTRY)')
  DeleteSQL.Strings = (
    'delete from EMPLOYEE '
    'where EMP_NO = :OLD_EMP_NO')
end
```

For new applications, you should consider using the IBDataSet component, which sums up the features of IBQuery and IBUpdateSQL. The differences between using the two components and the single component are minimal. Using IBQuery and IBUpdateSQL is a better approach when you're porting an existing application based on the two equivalent BDE components, even if porting the program directly to the IBDataSet component doesn't require much extra work.

In the IbxUpdSql example, I've provided both alternatives so you can test the differences yourself. Here is the skeleton of the DFM description of the single dataset component:

```
object IBDataSet1: TIBDataSet
  Database = IBDatabase1
  Transaction = IBTransaction1
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    '  (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, ' +
    '  JOB_COUNTRY)'
    'values'
    '  (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, ' +
    '  :JOB_GRADE, :JOB_COUNTRY)')
  SelectSQL.Strings = (...)
  UpdateRecordTypes = [cusUnmodified, cusModified, cusInserted]
  ModifySQL.Strings = (...)
end
```

If you connect the `IBQuery1` or the `IBDataSet1` component to the data source and run the program, you'll see that the behavior is identical. Not only do the components have a similar effect; the available properties and events are also similar.

In the IbxUpdSql program, I've also made the reference to the database a little more flexible. Instead of typing in the database name at design time, I've extracted the Borland shared data folder from the Windows Registry (where Borland saves it while installing Delphi). Here is the code executed when the program starts:

```
uses
  Registry;

procedure TForm1.FormCreate(Sender: TObject);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    Reg.OpenKey('\Software\Borland\Borland Shared\Data', False);
    IBDatabase1.DatabaseName := Reg.ReadString('Rootdir') + '\employee.gdb';
  finally
 Reg.Free;
  end;
  EmpDS.DataSet.Open;
end;
```

NOTE *For more information about the Windows Registry and INI files, see the related sidebar in Chapter 8, "The Architecture of Delphi Applications."*

Another feature of this example is the presence of a transaction component. As I've said, the InterBase Express components make the use of a transaction component compulsory, explicitly following a requirement of InterBase. Simply adding a couple of buttons to the form to commit or roll back the transaction would be enough, because a transaction starts automatically as you edit any dataset attached to it.

I've also improved the program by adding an ActionList component. This component includes all the standard database actions and adds two custom actions for transaction support: Commit and Rollback. Both actions are enabled when the transaction is active:

```
procedure TForm1.ActionUpdateTransactions(Sender: TObject);
begin
  acCommit.Enabled := IBTransaction1.InTransaction;
  acRollback.Enabled := acCommit.Enabled;
end;
```

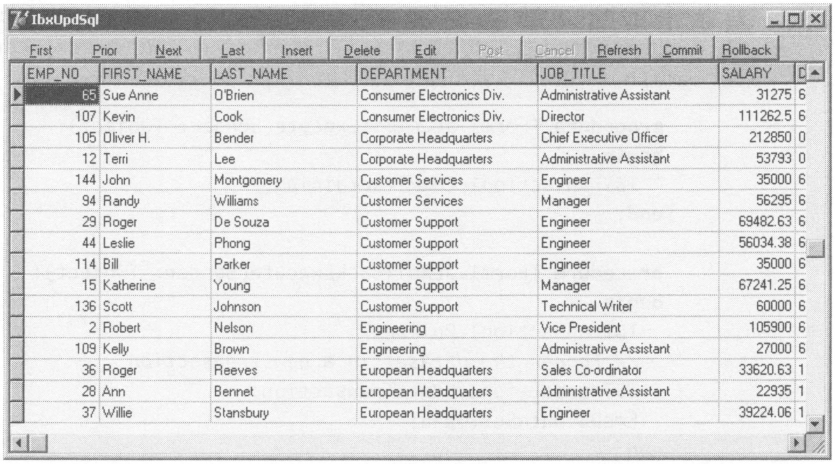
When executed, they perform the main operation but also need to reopen the dataset in a new transaction (which can also be done by "retaining" the transaction context). `CommitRetaining` doesn't really reopen a new transaction, but it allows the current transaction to remain open. This way, you can keep using your datasets, which won't be refreshed (so you won't see edits already committed by other users) but will keep showing the data you've modified. Here is the code:

```
procedure TForm1.acCommitExecute(Sender: TObject);
begin
  IBTransaction1.CommitRetaining;
end;

procedure TForm1.acRollbackExecute(Sender: TObject);
begin
  IBTransaction1.Rollback;
  // reopen the dataset in a new transaction
  IBTransaction1.StartTransaction;
  EmpDS.DataSet.Open;
end;
```

WARNING *Be aware that InterBase closes any opened cursors when a transaction ends, which means you have to reopen them and refetch the data even if you haven't made any changes. When committing data, however, you can ask InterBase to retain the* transaction context*—not to close open datasets—by issuing a* `CommitRetaining` *command, as mentioned before. InterBase behaves this way because a transaction corresponds to a snapshot of the data. Once a transaction is finished, you are supposed to read the data again to refetch records that may have been modified by other users. Version 6.0 of InterBase includes a* `RollbackRetaining` *command, but I've decided not to use it because in a rollback operation, the program should refresh the dataset data to show the original values on screen, not the updates you've discarded.*

The last operation refers to a generic dataset and not a specific one, because I'm going to add a second alternate dataset to the program. The actions are connected to a text-only toolbar, as you can see in Figure 14.14. The program opens the dataset at startup and automatically closes the current transaction on exit, after asking the user what to do, with the following `OnClose` event handler:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  nCode: Word;
begin
  if IBTransaction1.InTransaction then
  begin
    nCode := MessageDlg ('Commit Transaction? (No to rollback)',
      mtConfirmation, mbYesNoCancel, 0);
    case nCode of
      mrYes: IBTransaction1.Commit;
      mrNo: IBTransaction1.Rollback;
      mrCancel: Action := caNone; // don't close
    end;
  end;
end;
```



EMP_NO	FIRST_NAME	LAST_NAME	DEPARTMENT	JOB_TITLE	SALARY	D
65	Sue Anne	O'Brien	Consumer Electronics Div.	Administrative Assistant	31275	6
107	Kevin	Cook	Consumer Electronics Div.	Director	111262.5	6
105	Oliver H.	Bender	Corporate Headquarters	Chief Executive Officer	212850	0
12	Terri	Lee	Corporate Headquarters	Administrative Assistant	53793	0
144	John	Montgomery	Customer Services	Engineer	35000	6
94	Randy	Williams	Customer Services	Manager	56295	6
29	Roger	De Souza	Customer Support	Engineer	69482.63	6
44	Leslie	Phong	Customer Support	Engineer	56034.38	6
114	Bill	Parker	Customer Support	Engineer	35000	6
15	Katherine	Young	Customer Support	Manager	67241.25	6
136	Scott	Johnson	Customer Support	Technical Writer	60000	6
2	Robert	Nelson	Engineering	Vice President	105900	6
109	Kelly	Brown	Engineering	Administrative Assistant	27000	6
36	Roger	Reeves	European Headquarters	Sales Co-ordinator	33620.63	1
28	Ann	Bennet	European Headquarters	Administrative Assistant	22935	1
37	Willie	Stansbury	European Headquarters	Engineer	39224.06	1

FIGURE 14.14
The output of the IbxUpdSql example

Monitoring InterBase Express

Like the dbExpress architecture, IBX also allows you to monitor a connection. You can embed a copy of the IBSQLMonitor component in your application and produce a custom log.

You can even write a more generic monitoring application, as I've done in the IbxMon example. I've placed in its form a monitoring component and a RichEdit control, and written the following handler for the `OnSQL` event:

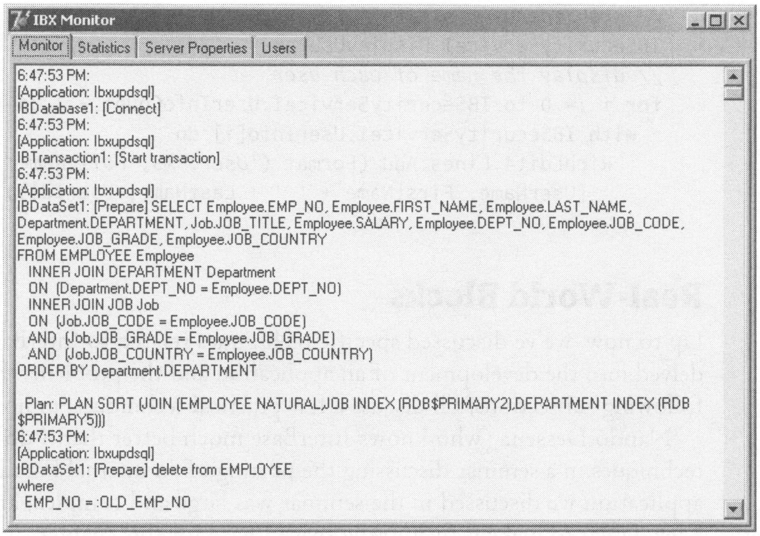
```
procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
  if Assigned (RichEdit1) then
    RichEdit1.Lines.Add (TimeToStr (Now) + ': ' + EventText);
end;
```

The `if Assigned` test can be useful when receiving a message during shutdown, and it is required when you add this code directly inside the application you are monitoring.

To receive messages from other applications (or from the current application), you have to turn on the IBDatabase component's tracing options. In the IbxUpdSql example (discussed in the preceding section, "Building a Live Query"), I turned them all on:

```
object IBDatabase1: TIBDatabase
  ...
  TraceFlags = [tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
                tfConnect, tfTransact, tfBlob, tfService, tfMisc]
```

If you run the two examples at the same time, the output of the IbxMon program will list details about the IbxUpdSql program's interaction with InterBase, as you can see in Figure 14.15.

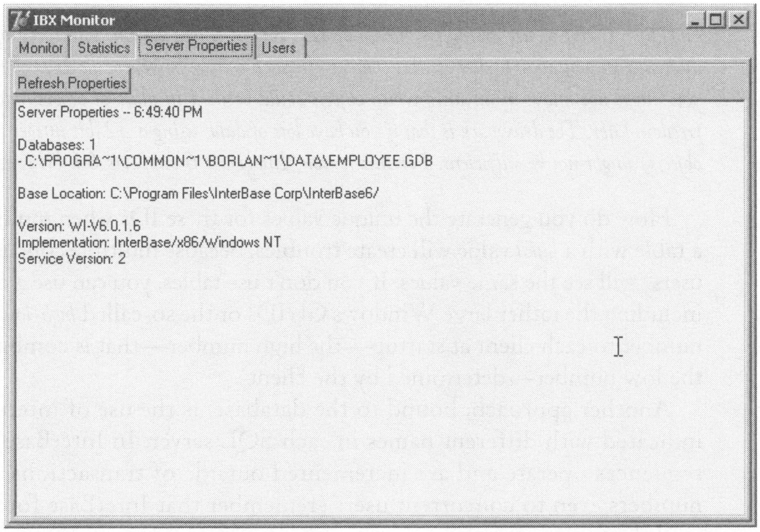


```
6:47:53 PM:
[Application: Ibxupdsql]
IBDatabase1: [Connect]
6:47:53 PM:
[Application: Ibxupdsql]
IBTransaction1: [Start transaction]
6:47:53 PM:
[Application: Ibxupdsql]
IBDataSet1: [Prepare] SELECT Employee.EMP_NO, Employee.FIRST_NAME, Employee.LAST_NAME,
Department.DEPARTMENT, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, Employee.JOB_CODE,
Employee.JOB_GRADE, Employee.JOB_COUNTRY
FROM EMPLOYEE Employee
  INNER JOIN DEPARTMENT Department
  ON  (Department.DEPT_NO = Employee.DEPT_NO)
  INNER JOIN JOB Job
  ON  (Job.JOB_CODE = Employee.JOB_CODE)
  AND  (Job.JOB_GRADE = Employee.JOB_GRADE)
  AND  (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)
ORDER BY Department.DEPARTMENT

 Plan: PLAN SORT (JOIN (EMPLOYEE NATURAL,JOB INDEX (RDB$PRIMARY2),DEPARTMENT INDEX (RDB
$PRIMARY5)))
6:47:53 PM:
[Application: Ibxupdsql]
IBDataSet1: [Prepare] delete from EMPLOYEE
where
 EMP_NO = :OLD_EMP_NO
```

FIGURE 14.15
The output of the IbxMon example, based on the IBMonitor component

Getting More System Data

In addition to letting you monitor the InterBase connection, the IbxMon example allows you to query some server settings using the various tabs on its page control. The example embeds a few IBX administrative components, showing server statistics, server properties, and all connected users. You can see an example of the server properties in Figure 14.16. The code for extracting the users appears in the following code fragment.



```
Server Properties -- 6:49:40 PM

Databases: 1
- C:\PROGRA~1\COMMON~1\BORLAN~1\DATA\EMPLOYEE.GDB

Base Location: C:\Program Files\InterBase Corp\InterBase6/

Version: WI-V6.0.1.6
Implementation: InterBase/x86/Windows NT
Service Version: 2
```

FIGURE 14.16
The server information displayed by the IbxMon application

```
// grab the user's data
IBSecurityService1.DisplayUsers;
// display the name of each user
for i := 0 to IBSecurityService1.UserInfoCount - 1 do
  with IBSecurityService1.UserInfo[i] do
    RichEdit4.Lines.Add (Format ('User: %s, Full Name: %s, Id: %d',
      [UserName, FirstName + ' ' + LastName, UserId]));
```

Real-World Blocks

Up to now, we've discussed specific techniques related to InterBase programming, but we haven't delved into the development of an application and the problems this presents in practice. In the following subsections, I'll discuss a few practical techniques, in no specific order.

Nando Dessena (who knows InterBase much better than I do) and I have used all of these techniques in a seminar discussing the porting of an internal Paradox application to InterBase. The application we discussed in the seminar was large and complex, and I've trimmed it down to only a few tables to make it fit into the space I have for this chapter.

TIP *The database discussed in this section is called* `mastering.gdb`*. You can find it in the* `data` *subfolder of the code folder for this chapter. You can examine it using InterBase Console, possibly after making a copy to a writable drive so that you can fully interact with it.*

Generators and IDs

I mentioned in Chapter 13 that I'm a fan of using IDs extensively to identify the records in each table of a database.

NOTE *I tend to use a single sequence of IDs for an entire system, something often called an Object ID (OID) and discussed in a sidebar earlier in this chapter. In such a case, however, the IDs of the two tables must be unique. Because you might not know in advance which objects could be used in place of others, adopting a global OID allows you more freedom later. The drawback is that if you have lots of data, using a 32-bit integer as the ID (that is, having only 4 billion objects) might not be sufficient. For this reason, InterBase 6 supports 64-bit generators.*

How do you generate the unique values for these IDs when multiple clients are running? Keeping a table with a *latest* value will create troubles, because multiple concurrent transactions (from different users) will see the same values. If you don't use tables, you can use a database-independent mechanism, including the rather large Windows GUIDs or the so-called *high-low technique* (the assignment of a base number to each client at startup—the high number—that is combined with a consecutive number—the low number—determined by the client).

Another approach, bound to the database, is the use of internal mechanisms for sequences, indicated with different names in each SQL server. In InterBase they are called *generators*. These sequences operate and are incremented outside of transactions, so that they provide unique numbers even to concurrent users (remember that InterBase forces you to open a transaction to read data).

You've already seen how to create a generator. Here is the definition for the one in my demo database, followed by the definition of the view you can use to query for a new value:

```
create generator g_master;

create view v_next_id (
  next_id
  ) as
select gen_id(g_master, 1) from rdb$database
;
```

Inside the RWBlocks application, I've added an IBQuery component to a data module (because I don't need it to be an editable dataset) with the following SQL:

```
select next_id from v_next_id;
```

The advantage, compared to using the direct statement, is that this code is easier to write and maintain, even if the underlying generator changes (or you switch to a different approach behind the scenes). Moreover, in the same data module, I've added a function that returns a new value for the generator:

```
function TDmMain.GetNewId: Integer;
begin
  // return the next value of the generator
  QueryId.Open;
  try
    Result := QueryId.Fields[0].AsInteger;
  finally
    QueryId.Close;
  end;
end;
```

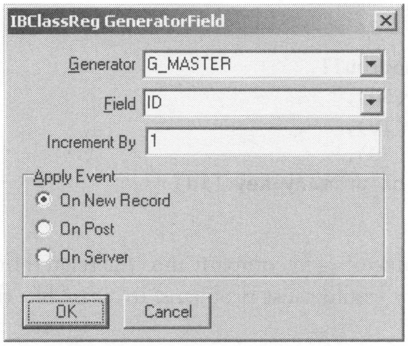
This method can be called in the `AfterInsert` event of any dataset to fill in the value for the ID:

```
mydataset.FieldByName ('ID').AsInteger := data.GetNewId;
```

As I've mentioned, the IBX datasets can be tied directly to a generator, thus simplifying the overall picture. Thanks to the specific property editor (shown in Figure 14.17), connecting a field of the dataset to the generator becomes trivial.

FIGURE 14.17

The editor for the `GeneratorField` property of the IBX datasets



IBClassReg GeneratorField

Generator: G_MASTER

Field: ID

Increment By: 1

Apply Event

● On New Record

○ On Post

○ On Server

Notice that both these approaches are much better than the approach based on a server-side trigger, discussed earlier in this chapter. In that case, the Delphi application didn't know the ID of the record sent to the database and so was unable to refresh it. Not having the record ID (which is also the only key field) on the Delphi side means it is almost impossible to insert such a value directly inside a DBGrid. If you try, you'll see that the value you insert gets lost, only to reappear in case of a full refresh.

Using client-side techniques based on the manual code or the `GeneratorField` property causes no trouble. The Delphi application knows the ID (the record key) before posting it, so it can easily place it in a grid and refresh it properly.

Case-Insensitive Searches

An interesting issue with SQL servers in general, not specifically InterBase, has to do with case-insensitive searches. Suppose you don't want to show a large amount of data in a grid (which is a bad idea for a client/server application). You instead choose to let the user type the initial portion of a name and then filter a query on this input, displaying only the smaller resulting record set in a grid. I've done this for a table of companies.

This search by company name will be executed frequently and will take place on a large table. However, if you search using the `starting with` or `like` operator, the search will be case sensitive, as in the following SQL statement:

```
select * from companies
where name starting with 'win';
```

To make a case-insensitive search, you can use the `upper` function on both sides of the comparison to test the uppercase values of each string, but a similar query will be very slow, because it won't be based on an index. On the other hand, saving the company names (or any other name) in uppercase letters would be silly, because when you print those names, the result will be unnatural (even if common in old information systems).

If you can trade off some disk space and memory for the extra speed, you can use a trick: Add an extra field to the table to store the uppercase value of the company name, and use a server-side trigger to generate it and update it. You can then ask the database to maintain an index on the uppercase version of the name, to speed the search operation even further.

In practice, the table definition looks like this:

```
create domain d_uid as integer;
create table companies
(
  id          d_uid not null,
  name        varchar(50),
  tax_code    varchar(16),
  name_upper  varchar(50),
constraint companies_pk primary key (id)
);
```

To copy the uppercase name of each company into the related field, you cannot rely on client-side code, because an inconsistency would cause problems. In a case like this, it is better to use a trigger on the server, so that each time the company name changes, its uppercase version is updated accordingly. Another trigger is used to insert a new company:

```
create trigger companies_bi for companies
active before insert position 0
as
begin
  new.name_upper = upper(new.name);
end;

create trigger companies_bu for companies
active before update position 0
as
begin
  if (new.name <> old.name) then
    new.name_upper = upper(new.name);
end;
```

Finally, I've added an index to the table with this DDL statement:

```
create index i_companies_name_upper on companies(name_upper);
```

With this structure behind the scenes, you can now select all the companies starting with the text of an edit box (edSearch) by writing the following code in a Delphi application:

```
dm.DataCompanies.Close;
dm.DataCompanies.SelectSQL.Text :=
  'select c.id, c.name, c.tax_code,' +
  '  from companies c ' +
  '  where name_upper starting with ''' +
  UpperCase (edSearch.Text) + '''';
dm.DataCompanies.Open;
```

TIP *Using a prepared parametric query, you might be able to make this code even faster.*

As an alternative, you could create a server-side calculated field in the table definition, but doing so would prevent you from having an index on the field, which speeds up your queries considerably:

```
name_upper  varchar(50) computed by (upper(name))
```

Handling Locations and People

You might notice that the table describing companies is quite bare. It has no company address, nor any contact information. The reason is that I want to be able to handle companies that have multiple offices (or locations) and list contact information about multiple employees of those companies.

Every location is bound to a company. Notice, though, that I've decided not to use a location identifier related to the company (such as a progressive location number for each company), but rather a global ID for all the locations. This way, I can refer to a location ID (let's say, for shipping goods) without having to also refer to the company ID. This is the definition of the table that stores company locations:

```
create table locations
(
  id         d_uid not null,
  id_company d_uid not null,
  address    varchar(40),
  town       varchar(30),
  zip        varchar(10),
  state      varchar(4),
  phone      varchar(15),
  fax        varchar(15),
constraint locations_pk primary key (id),
constraint locations_uc unique (id_company, id)
);

alter table locations add constraint locations_fk_companies
  foreign key (id_company) references companies (id)
  on update no action on delete no action;
```

The final definition of a foreign key relates the id_company field of the locations table with the ID field of the companies table. The other table lists names and contact information for people at specific company locations. To follow the database normalization rules, I should have added to this table only a reference to the location, because each location relates to a company. However, to make it simpler to change the location of a person within a company and to make my queries much more efficient (avoiding an extra step), I've added to the people table both a reference to the location and a reference to the company.

The table has another unusual feature: One of the people working for a company can be set as the key contact. You obtain this functionality with a Boolean field (defined with a domain, because the Boolean type is not supported by InterBase) and by adding triggers to the table so that only one employee of each company can have this flag active:

```
create domain d_boolean as char(1)
  default 'F'
  check (value in ('T', 'F')) not null

create table people
(
  id          d_uid not null,
  id_company  d_uid not null,
  id_location d_uid not null,
  name        varchar(50) not null,
  phone       varchar(15),
  fax         varchar(15),
  email       varchar(50),
  key_contact d_boolean,
constraint people_pk primary key (id),
constraint people_uc unique (id_company, name)
);

alter table people add constraint people_fk_companies
  foreign key (id_company) references companies (id)
  on update no action on delete cascade;
alter table people add constraint people_fk_locations
  foreign key (id_company, id_location)
  references locations (id_company, id);

create trigger people_ai for people
active after insert position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;

create trigger people_au for people
active after update position 0
as
begin
  /* if a person is the key contact, remove the
     flag from all others (of the same company) */
  if (new.key_contact = 'T' and old.key_contact = 'F') then
    update people
    set key_contact = 'F'
    where id_company = new.id_company
    and id <> new.id;
end;
```

Building a User Interface

The three tables discussed so far have a clear master/detail relation. For this reason, the RWBlocks example uses three IBDataSet components to access the data, hooking up the two secondary tables to the main one. The code for the master/detail support is that of a standard database example based on queries, so I won't discuss it further (but I suggest you study the example's source code).

Each of the datasets has a full set of SQL statements, to make the data editable. Whenever you enter a new detail element, the program hooks it to its master tables, as in the two following methods:

```
procedure TDmCompanies.DataLocationsAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataLocationsID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
end;

procedure TDmCompanies.DataPeopleAfterInsert(DataSet: TDataSet);
begin
  // initialize the data of the detail record
  // with a reference to the master record
  DataPeopleID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
  // the suggested location is the active one, if available
  if not DataLocations.IsEmpty then
    DataPeopleID_LOCATION.AsInteger := DataLocationsID.AsInteger;
  // the first person added becomes the key contact
  // (checks whether the filtered dataset of people is empty)
  DataPeopleKEY_CONTACT.AsBoolean := DataPeople.IsEmpty;
end;
```

As this code suggests, a data module hosts the dataset components. The program has a data module for every form (hooked up dynamically, because you can create multiple instances of each form). Each data module has a separate transaction so that the various operations performed in different pages are totally independent. The database connection, however, is centralized. A main data module hosts the corresponding component, which is referenced by all the datasets. Each of the data modules is created dynamically by the form referring to it, and its value is stored in the form's dm private field:

```
procedure TFormCompanies.FormCreate(Sender: TObject);
begin
  dm := TDmCompanies.Create (Self);
  dsCompanies.Dataset := dm.DataCompanies;
  dsLocations.Dataset := dm.DataLocations;
  dsPeople.Dataset := dm.DataPeople;
end;
```

This way, you can easily create multiple instances of a form, with an instance of the data module connected to each of them. The form connected to the data module has three DBGrid controls, each tied to a data module and one of the corresponding datasets. You can see this form at run time, with some data, in Figure 14.18.

The form is hosted by a main form, which in turn is based on a page control, with the other forms embedded. Only the form connected with the first page is created when the program starts. The `ShowForm` method I've written takes care of parenting the form to the tab sheet of the page control, after removing the form border:

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  ShortDateFormat := 'dd/mm/yyyy';
  ShowForm (TFormCompanies.Create (Self), TabCompanies);
end;
```

```
procedure TFormMain.ShowForm (Form: TForm; Tab: TTabSheet);
begin
  Form.BorderStyle := bsNone;
  Form.Align := alClient;
  Form.Parent := Tab;
  Form.Show;
end;
```

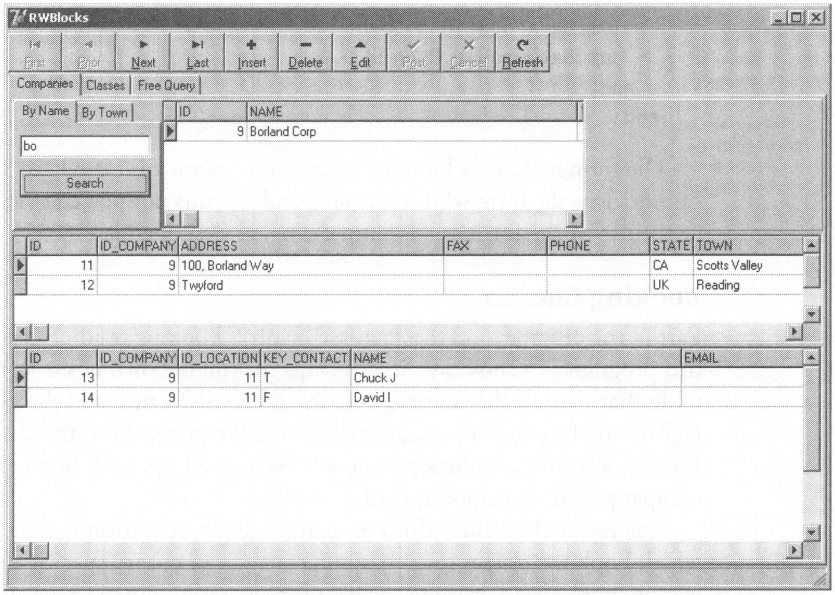


FIGURE 14.18

A form showing companies, office locations, and people (part of the RWBlocks example)

RWBlocks
First
Prior
Next
Last
Insert
Delete
Edit
Post
Cancel
Refresh
Companies
Classes
Free Query
By Name
By Town
Search

bo

ID	NAME
9	Borland Corp

ID	ID_COMPANY	ADDRESS	FAX	PHONE	STATE	TOWN
11	9	100, Borland Way			CA	Scotts Valley
12	9	Twyford			UK	Reading

ID	ID_COMPANY	ID_LOCATION	KEY_CONTACT	NAME	EMAIL
13	9	11	T	Chuck J	
14	9	11	F	David I	

The other two pages are populated at run time:

```
procedure TFormMain.PageControl1Change(Sender: TObject);
begin
  if PageControl1.ActivePage.ControlCount = 0 then
    if PageControl1.ActivePage = TabFreeQ then
      ShowForm (TFormFreeQuery.Create (self), TabFreeQ)
    else if PageControl1.ActivePage = TabClasses then
      ShowForm (TFormClasses.Create (self), TabClasses);
end;
```

The companies form hosts the search by company name (discussed in the previous section) plus a search by location. You enter the name of a town and get back a list of companies having an office in that town:

```
procedure TFormCompanies.btnTownClick(Sender: TObject);
begin
  with dm.DataCompanies do
```

```
    begin
      Close;
      SelectSQL.Text :=
        'select c.id, c.name, c.tax_code' +
        '  from companies c ' +
        '  where exists (select loc.id from locations loc ' +
        '  where loc.id_company = c.id and upper(loc.town) = ''' +
        UpperCase(edTown.Text) + ''' )';
      Open;
      dm.DataLocations.Open;
      dm.DataPeople.Open;
    end;
  end;
```

The form includes a lot more source code. Some of it is related to closing permission (as a user cannot close the form while there are pending edits not posted to the database), and quite a bit relates to the use of the form as a lookup dialog, as described later.

Booking Classes

Part of the program and the database involves booking training classes and courses. (Although I built this program as a showcase, it also helps me run my own business.) The database includes a classes table that lists all the training courses, each with a title and the planned date. Another table hosts registration by company, including the classes registered for, the ID of the company, and some notes. Finally, a third table lists people who've signed up, each hooked to a registration for his or her company, with the amount paid.

The rationale behind this company-based registration is that invoices are sent to companies, which book the classes for programmers and can receive specific discounts. In this case the database is more normalized, because the people registration doesn't refer directly to a class, but only to the company registration for that class. Here are the definitions of the tables involved (I've omitted foreign key constraints and other elements):

```
create table classes
(
  id           d_uid not null,
  description  varchar(50),
  starts_on   timestamp not null,
constraint classes_pk primary key (id)
);
create table classes_reg
(
  id          d_uid not null,
  id_company  d_uid not null,
  id_class    d_uid not null,
  notes       varchar(255),
constraint classes_reg_pk primary key (id),
constraint classes_reg_uc unique (id_company, id_class)
);
create domain d_amount as numeric(15, 2);
create table people_reg
(
  id             d_uid not null,
  id_classes_reg d_uid not null,
  id_person      d_uid not null,
  amount         d_amount,
constraint people_reg_pk primary key (id)
);
```

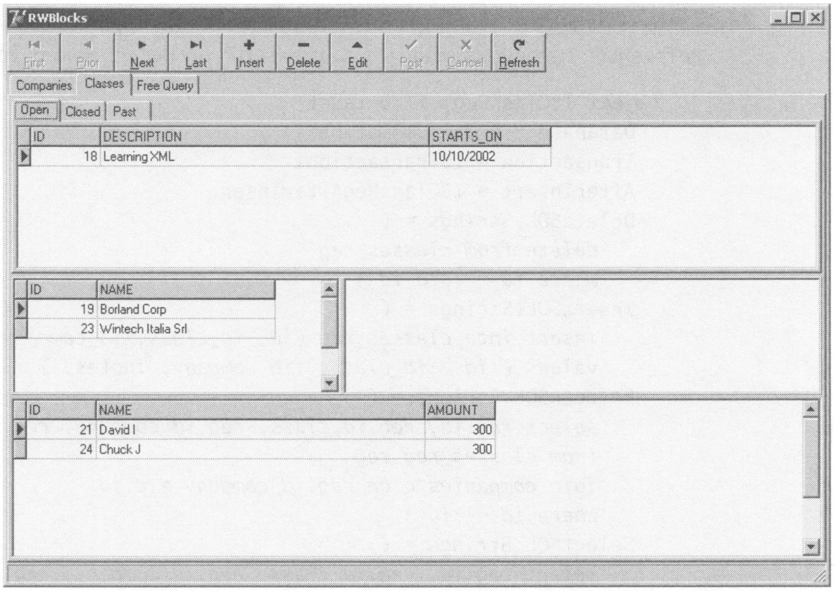
The data module for this group of tables uses a master/detail/detail relationship, and has code to set the connection with the active master record when a new detail record is created. Each dataset has a generator field for its ID, and each has the proper `update` and `insert` SQL statements. These statements are generated by the corresponding component editor using only the ID field to identify existing records and updating only the fields in the original table. Each of the two secondary datasets retrieves data from a lookup table (either the list of companies or the list of people). I had to edit the `RefreshSQL` statements manually to repeat the proper inner join. Here is an example:

```
object IBClassReg: TIBDataSet
  Database = DmMain.IBDatabase1
  Transaction = IBTransaction1
  AfterInsert = IBClassRegAfterInsert
  DeleteSQL.Strings = (
    'delete from classes_reg'
    'where id = :old_id')
  InsertSQL.Strings = (
    'insert into classes_reg (id, id_class, id_company, notes)'
    'values (:id, :id_class, :id_company, :notes)')
  RefreshSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id = :id')
  SelectSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id_class = :id')
  ModifySQL.Strings = (
    'update classes_reg'
    'set'
    '  id = :id,'
    '  id_class = :id_class,'
    '  id_company = :id_company,'
    '  notes = :notes'
    'where id = :old_id')
  GeneratorField.Field = 'id'
  GeneratorField.Generator = 'g_master'
  DataSource = dsClasses
end
```

To complete the discussion of `IBClassReg`, here is its only event handler:

```
procedure TDmClasses.IBClassRegAfterInsert(DataSet: TDataSet);
begin
  IBClassReg.FieldByName ('id_class').AsString :=
    IBClasses.FieldByName ('id').AsString;
end;
```

The `IBPeopleReg` dataset has similar settings, but the `IBClasses` dataset is simpler at design time. At run time, this dataset's SQL code is dynamically modified, using three alternatives to display scheduled classes (whenever the date is after today's date), classes already started or finished in the current year, and classes from past years. A user selects one of the three groups of records for the table with a tab control, which hosts the DBGrid for the main table (see Figure 14.19).



ID	DESCRIPTION	STARTS_ON
18	Learning XML	10/10/2002

ID	NAME
19	Borland Corp
23	Wintech Italia Srl

ID	NAME	AMOUNT
21	David I	300
24	Chuck J	300

FIGURE 14.19
The RWBlocks example form for class registrations

The three alternative SQL statements are created when the program starts, or when the class registrations form is created and displayed. The program stores the final portion of the three alternative instructions (the `where` clause) in a string list and selects one of the strings when the tab changes:

```
procedure TFormClasses.FormCreate(Sender: TObject);
begin
  dm := TDmClasses.Create (Self);
  // connect the datasets to the data sources
  dsClasses.Dataset := dm.IBClasses;
  dsClassReg.DataSet := dm.IBClassReg;
  dsPeopleReg.DataSet := dm.IBPeopleReg;
  // open the datasets
  dm.IBClasses.Active := True;
```

```
    dm.IBClassReg.Active := True;
    dm.IBPeopleReg.Active := True;

    // prepare the SQL for the three tabs
    SqlCommands := TStringList.Create;
    SqlCommands.Add (' where Starts_On > ''now''');
    SqlCommands.Add (' where Starts_On <= ''now'' and ' +
      ' extract (year from Starts_On ) >= extract(year from current_timestamp)');
    SqlCommands.Add (' where extract (year from Starts_On) < ' +
      ' extract(year from current_timestamp)');
  end;
  procedure TFormClasses.TabChange(Sender: TObject);
  begin
    dm.IBClasses.Active := False;
    dm.IBClasses.SelectSQL [1] := SqlCommands [Tab.TabIndex];
    dm.IBClasses.Active := True;
  end;
```

Building a Lookup Dialog

The two detail datasets of this class registration form display lookup fields. Instead of showing the ID of the company that booked the class, for example, the form shows the company name. You obtain this functionality with an inner join in the SQL statement and by configuring the DBGrid columns so they don't display the company ID. In a local application, or one with a limited amount of data, you could use a lookup field. However, copying the entire lookup dataset locally or opening it for browsing should be limited to tables with about 100 records at most, embedding some search capabilities.

If you have a large table, such as a table of companies, an alternative solution is to use a secondary dialog box to perform the lookup selection. For example, you can choose a company by using the form you've already built and taking advantage of its search capabilities. To display this form as a dialog box, the program creates a new instance of it, shows some hidden buttons already there at design time, and lets the user select a company to refer to from the other table.

To simplify the use of this lookup, which can happen multiple times in a large program, I've added to the companies form a class function that has as output parameters the name and ID of the selected company. An initial ID can be passed to the function to determine its initial selection. Here is the complete code of this class function, which creates an object of its class, selects the initial record if requested, shows the dialog box, and finally extracts the return values:

```
  class function TFormCompanies.SelectCompany (
    var CompanyName: string; var CompanyId: Integer): Boolean;
  var
    FormComp: TFormCompanies;
  begin
    Result := False;
    FormComp := TFormCompanies.Create (Application);
    FormComp.Caption := 'Select Company';
  try
    // activate dialog buttons
    FormComp.btnCancel.Visible := True;
    FormComp.btnOK.Visible := True;
    // select company
    if CompanyId > 0 then
      FormComp.dm.DataCompanies.SelectSQL.Text :=
        'select c.id, c.name, c.tax_code' +
        '  from companies c ' +
        '  where c.id = ' + IntToStr (CompanyId)
    else
      FormComp.dm.DataCompanies.SelectSQL.Text :=
        'select c.id, c.name, c.tax_code' +
        '  from companies c ' +
        '  where name_upper starting with ''a''';
    FormComp.dm.DataCompanies.Open;
    FormComp.dm.DataLocations.Open;
    FormComp.dm.DataPeople.Open;

    if FormComp.ShowModal = mrOK then
    begin
      Result := True;
      CompanyId := FormComp.dm.DataCompanies.FieldByName ('id').AsInteger;
      CompanyName := FormComp.dm.DataCompanies.FieldByName ('name').AsString;
    end;
  finally
    FormComp.Free;
  end;
end;
```

Another slightly more complex class function (available with the example's source code, but not listed here) lets you select a person from a given company to register people for classes. In this case, the form is displayed after disallowing searching another company or modifying the company's data.

In both cases, you trigger the lookup by adding an ellipsis button to the column of the DBGrid—for example, the grid column listing the names of companies registered for classes. When this button is clicked, the program calls the class function to display the dialog box and uses its result to update the hidden ID field and the visible name field:

```
procedure TFormClasses.DBGridClassRegEditButtonClick(Sender: TObject);
var
  CompanyName: string;
  CompanyId: Integer;
begin
  CompanyId := dm.IBClassReg.FieldByName ('id_Company').AsInteger;
  if TFormCompanies.SelectCompany (CompanyName, CompanyId) then
  begin
    dm.IBClassReg.Edit;
    dm.IBClassReg.FieldByName ('Name').AsString := CompanyName;
    dm.IBClassReg.FieldByName ('id_Company').AsInteger := CompanyId;
  end;
end;
```

Adding a Free Query Form

The program's final feature is a form where a user can directly type in and run a SQL statement. As a helper, the form lists in a combo box the available tables of the database, obtained when the form is created by calling

```
DmMain.IBDatabase1.GetTableNames (ComboTables.Items);
```

Selecting an item from the combo box generates a generic SQL query:

```
MemoSql.Lines.Text := 'select * from ' + ComboTables.Text;
```

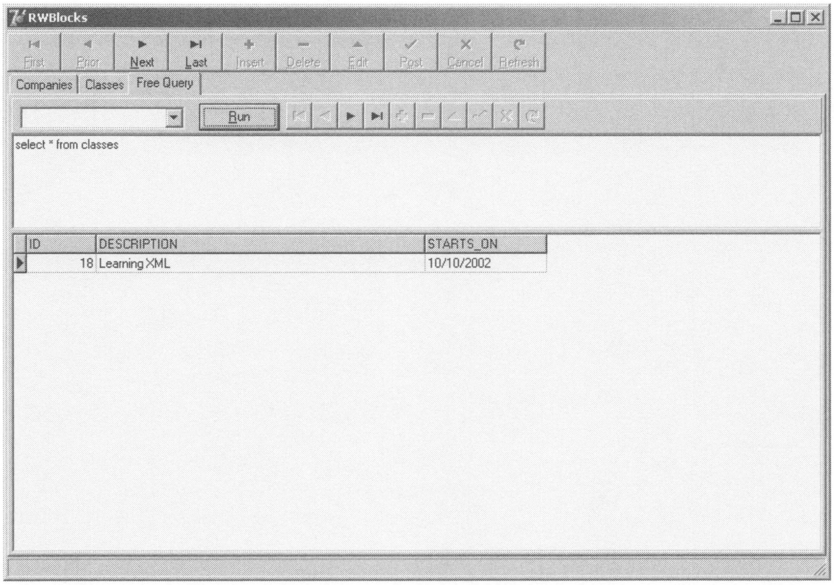
The user (if an expert) can then edit the SQL, possibly introducing restrictive clauses, and then run the query:

```
procedure TFormFreeQuery.ButtonRunClick(Sender: TObject);
begin
  QueryFree.Close;
  QueryFree.SQL := MemoSql.Lines;
  QueryFree.Open;
end;
```

You can see this third form of the RWBlocks program in Figure 14.20. Of course, I'm not suggesting that you add SQL editing to programs intended for all your users—this feature is intended for power users or programmers. I basically wrote it for myself!

FIGURE 14.20

The free query form of the RWBlocks example is intended for power users.



select * from classes

ID	DESCRIPTION	STARTS_ON
18	Learning XML	10/10/2002

What's Next?

This chapter has presented a detailed introduction to client/server programming with Delphi. We discussed the key issues and delved a little into interesting areas of client/server programming. After a general introduction, I discussed the use of the dbExpress database library. I also briefly covered the InterBase server and the InterBase Express (IBX) components. At the end of the chapter I presented a real-world example, going beyond the typical examples in the book to focus on a single feature at a time.

There is more I can say about client/server programming in Delphi. Chapter 15 will focus on Microsoft's ADO database engine. It's followed by a chapter on Delphi's multitier architecture, DataSnap; this chapter highlights the use of the dbExpress library and the ClientDataSet component, although in a slightly different context.

Chapter 15

Working with ADO

SINCE THE MID-1980S, database programmers have been on a quest for the "holy grail" of *database independence*. The idea is to use a single API that applications can use to interact with many different sources of data. The use of such an API would release developers from dependence on a single database engine and allow them to adapt to the world's changing demands. Vendors have produced many solutions to this goal, the two most notable early solutions being Microsoft's Open Database Connectivity (ODBC) and Borland's Integrated Database Application Programming Interface (IDAPI), more commonly known as the Borland Database Engine (BDE).

Microsoft started to replace ODBC with OLE DB in the mid-1990s with the success of the Component Object Model (COM). However, OLE DB is what Microsoft would class a *system-level* interface and is intended to be used by system-level programmers. It is very large, complex, and unforgiving. It makes greater demands on the programmer and requires a higher level of knowledge in return for lower productivity. ActiveX Data Objects (ADO) is a layer on top of OLE DB and is referred to as an *application-level* interface. It is considerably simpler than OLE DB and more forgiving. In short, it is designed for use by application programmers.

As you saw in Chapter 14, "Client/Server with dbExpress," Borland has also replaced the BDE with a newer technology called dbExpress. ADO shares greater similarities with the BDE than with the lightweight dbExpress technology. BDE and ADO support navigation and manipulation of datasets, transaction processing, and cached updates (called *batch updates* in ADO), so the concepts and issues involved in using ADO are similar to those of the BDE.

NOTE *I wish to acknowledge and thank Guy Smith-Ferrier for originally writing this chapter for* Mastering Delphi 6. *Guy is a programmer, author, and speaker. He is the author of several commercial software products and countless internal systems for independent and blue-chip companies alike. He has written many articles for* The Delphi Magazine *and for others on topics beyond Delphi, and he has spoken at numerous conferences in North America and Europe. Guy lives in England with his wife, his son, and his cat.*

In this chapter we will look at ADO. We will also discuss *dbGo*—a set of Delphi components initially called ADOExpress, but renamed in Delphi 6 because Microsoft objects to the use of the term *ADO* in third-party product names. It is possible to use ADO in Delphi without using dbGo. By importing the ADO type library, you can gain direct access to the ADO interfaces; this is how Delphi programmers used ADO before the release of Delphi 5. However, this path bypasses Delphi's database infrastructure and ensures that you are unable to make use of other Delphi technologies such as the data-aware controls or DataSnap. This chapter uses dbGo for all of its examples, not only because it is readily available and supported but also because it is a very viable solution. Regardless of your final choice, you will find the information here useful.

NOTE *Alternatively, you can turn to Delphi's active third-party market for other ADO component suites such as Adonis, AdoSolutio, Diamond ADO, and Kamiak.*

This chapter covers the following:

- Microsoft Data Access Components (MDAC)
- Delphi's dbGo
- Data link files
- Getting schema information
- Using the Jet engine
- Transaction processing
- Disconnected and persistent recordsets
- The briefcase model and deploying MDAC

Microsoft Data Access Components (MDAC)

ADO is part of a bigger picture called Microsoft Data Access Components (MDAC). MDAC is an umbrella for Microsoft's database technologies and includes ADO, OLE DB, ODBC, and RDS (Remote Data Services). Often you will hear people use the terms MDAC and ADO interchangeably, but incorrectly. Because ADO is only distributed as part of MDAC, we talk of ADO versions in terms of MDAC releases. The major releases of MDAC have been versions 1.5, 2.0, 2.1, 2.5, and 2.6. Microsoft releases MDAC independently and makes it available for free download and virtually free distribution (there are distribution requirements, but most Delphi developers will have no trouble meeting them). MDAC is also distributed with most Microsoft products that have database content. Delphi 7 ships with MDAC 2.6.

There are two consequences of this level of availability. First, it is highly likely that your users will already have MDAC installed on their machines. Second, whatever version your users have, or you upgrade them to, it is also virtually certain that someone—you, your users, or other application software—will upgrade their existing MDAC to the current release of MDAC. You can't prevent this upgrade, because MDAC is installed with such commonly used software as Internet Explorer. Add to this the fact that Microsoft supports only the current release of MDAC and the release before it, and you are arrive at this conclusion: Applications must be designed to work with the current release of MDAC or the release before it.

As an ADO developer, you should regularly check the MDAC pages on Microsoft's website at `www.microsoft.com/data`. From there you can download the latest version of MDAC for free. While you are on this website, you should take the opportunity to download the MDAC SDK (13 MB) if you do not already have it or the Platform SDK (the MDAC SDK is part of the Platform SDK). The MDAC SDK is your bible: Download it, consult it regularly, and use it to answer your ADO questions. You should treat it as your first port of call when you need MDAC information.

OLE DB Providers

OLE DB providers enable access to a source of data. They are ADO's equivalent to the dbExpress drivers and the BDE SQL Links. When you install MDAC, you automatically install the OLE DB providers shown in Table 15.1:

TABLE 15.1: OLE DB PROVIDERS INCLUDED WITH MDAC

DRIVER	PROVIDER	DESCRIPTION
MSDASQL	ODBC Drivers	ODBC drivers (default)
Microsoft.Jet.OLEDB.3.5	Jet 3.5	MS Access 97 databases only
Microsoft.Jet.OLEDB.4.0	Jet 4.0	MS Access and other databases
SQLOLEDB	SQL Server	MS SQL Server databases
MSDAORA	Oracle	Oracle databases
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	Example of an OLE DB provider for CSV files
MSDAOSP	Simple provider	For creating your own providers for simple text data

- The ODBC OLE DB provider is used for backward compatibility with ODBC. As you learn more about ADO, you will discover the limitations of this provider.
- The Jet OLE DB providers support MS Access and other desktop databases. We will return to these providers later.
- The SQL Server provider supports SQL Server 7, SQL Server 2000, and Microsoft Database Engine (MSDE). MSDE is a reduced version of SQL Server, with most of the tools removed and some code added to deliberately degrade performance when there are more than five active connections. MSDE is important because it is free and it is fully compatible with SQL Server.
- The OLE DB provider for OLAP can be used directly but is more often used by ADO Multi-Dimensional (ADOMD). ADOMD is an additional ADO technology designed to provide Online Analytical Processing (OLAP). If you have used Delphi's Decision Cube, Excel's Pivot Tables, or Access's Cross Tabs, then you have used some form of OLAP.

In addition to these MDAC OLE DB providers, Microsoft supplies other OLE DB providers with other products or with downloadable SDKs:

- The Active Directory Services OLE DB provider is included with the ADSI SDK; the AS/400 and VSAM OLE DB provider is included with SNA Server; and the Exchange OLE DB provider is included with Microsoft Exchange 2000.

- The OLE DB provider for Indexing Service is part of Microsoft Indexing Service, a Windows mechanism that speeds up file searches by building catalogs of file information. Indexing Service is integrated into IIS and, consequently, is often used for indexing websites.
- The OLE DB provider for Internet Publishing allows developers to manipulate directories and files using HTTP.
- Still more OLE DB providers come in the form of *service providers*. As their name implies, OLE DB service providers provide a service to other OLE DB providers and are often invoked automatically as needed without programmer intervention. The Cursor Service, for example, is invoked when you create a client-side cursor, and the Persisted Recordset provider is invoked to save data locally.

MDAC includes many providers that I'll discuss, but many more are available from Microsoft and from the third-party market. It is impossible to reliably list all available OLE DB providers, because the list is so large and changes constantly. In addition to independent third parties, you should consider most database vendors, because the majority now supply their own OLE DB providers. For example, Oracle supplies the ORAOLEDB provider.

TIP *A notable omission from the vendors that supply OLE DB providers is InterBase. In addition to accessing it using the ODBC driver, you can use Dmitry Kovalenko's IBProvider* (`www.lcpi.lipetsk.ru/prog/eng/index.html`)*. Also check Binh Ly's OLE DB Provider Development Toolkit* (`www.techvanguards.com/products/optk/install.htm`)*. If you want to write your own OLE DB provider, this tool is easier to use than most.*

Using dbGo Components

Programmers familiar with the BDE, dbExpress, or IBExpress should recognize the set of components that make up dbGo (Table 15.2).

TABLE 15.2: DBGO COMPONENTS

DBGO COMPONENT	DESCRIPTION	BDE EQUIVALENT COMPONENT
ADOConnection	Connection to a database	Database
ADOCommand	Executes an action SQL command	No equivalent
ADODataSet	All-purpose descendant of TDataSet	No equivalent
ADOTable	Encapsulation of a table	Table
ADOQuery	Encapsulation of SQL SELECT	Query
ADOStoredProc	Encapsulation of a stored procedure	StoredProc
RDSConnection	Remote Data Services connection	No equivalent

The four dataset components (ADODataSet, ADOTable, ADOQuery, and ADOStoredProc) are implemented almost entirely by their immediate ancestor class, TCustomADODataSet. This component provides the majority of dataset functionality, and its descendants are mostly thin wrappers that expose different features of the same component. As such, the components have a lot in common. In general, however, ADOTable, ADOQuery, and ADOStoredProc are viewed as "compatibility" components and are used to aid the transition of knowledge and code from their BDE counterparts. Be warned, though: These compatibility components are similar to their counterparts but not identical. You will find differences in any application except the most trivial. ADODataSet is the component of choice partly because of its versatility but also because it is closer in appearance to the ADO Recordset interface upon which it is based. Throughout this chapter, I'll use all the dataset components to give you the experience of using each.

A Practical Example

Enough theory: Let's see some action. Drop an ADOTable onto a form. To indicate the database to connect to, ADO uses *connection strings*. You can type in a connection string by hand if you know what you are doing. In general, you'll use the connection string editor (the property editor of the `ConnectionString` property), shown in Figure 15.1.

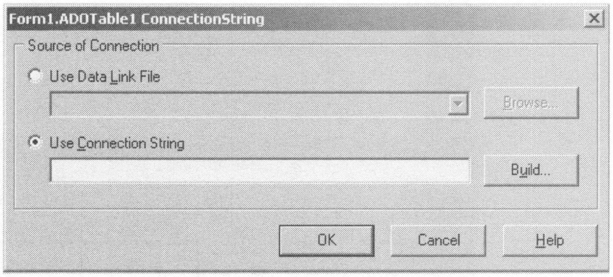


FIGURE 15.1

Delphi's connection string editor

This editor adds little value to the process of entering a connection string, so you can click Build to go straight to Microsoft's connection string editor, shown in Figure 15.2.

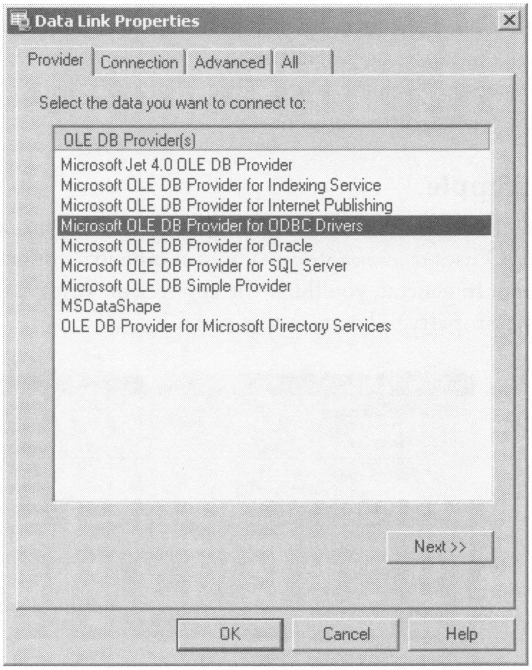
This is a tool you need to understand. The first tab shows the OLE DB providers and service providers installed on your computer. The list will vary according to the version of MDAC and other software you have installed. In this example, select the Jet 4.0 OLE DB provider. Double-click Jet 4.0 OLE DB Provider, and you will be presented with the Connection tab. This page varies according to the provider you select; for Jet, it asks you for the name of the database and your login details. You can choose an Access MDB file installed by Borland with Delphi 7: the `dbdemos.mdb` file available in the shared data folder (by default, `C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb`). Click the Test Connection button to test the validity of your selections.

The Advanced tab handles access control to the database; here you specify exclusive or read-only access to the database. The All tab lists all the parameters in the connection string. The list is specific to the OLE DB provider you selected on the first page. (You should make a mental note of this page, because it contains many parameters that are the answers to many problems.) After closing the Microsoft connection string editor you'll see in the Borland `ConnectionString` property editor the value that will be returned to the `ConnectionString` property (here split on multiple lines for readability):

```
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb;
Persist Security Info=False
```

FIGURE 15.2

The first page of Microsoft's connection string editor



Select the data you want to connect to:

OLE DB Provider(s)
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services

Connection strings are just strings with many parameters delimited by semicolons. To add, edit, or delete any of these parameter values programmatically, you must write your own routines to find the parameter in the list and amend it appropriately. A simpler approach is to copy the string into a Delphi string list and use its name/value pairs capability: This technique will be demonstrated in the JetText example covered in the section "Text Files through Jet."

Now that you have set the connection string, you can select a table. Drop down the list of tables using the `TableName` property in the Object Inspector. Select the Customer table. Add a DataSource component and a DBGrid control and connect them all together; you are now using ADO in an actual—though trivial—program (available in the source code as FirstAdoExample). To see the data, set the `Active` property of the dataset to True or open the dataset in the `FormCreate` event (as in the example) to avoid design-time errors if the database is not available.

TIP *If you are going to use dbGo as your primary database access technology, you might want to move the DataSource component to the ADO page of the Component Palette to avoid moving back and forth between the ADO page and the Data Access page. If you use both ADO and another database technology, then you can simulate installing DataSource on multiple pages by creating a Component Template for a DataSource and installing it on the ADO page.*

The ADOConnection Component

When you use an ADOTable component this way, it creates its own connection component behind the scenes. You do not have to accept the default connection it creates. In general, you should create your own connection using the ADOConnection component, which has the same purpose as the dbExpress SQLConnection component and the BDE Database component. It allows you to customize the login procedure, control transactions, execute action commands directly, and reduce the number of connections in an application.

Using an ADOConnection is easy. Place one on a form and set its `ConnectionString` property the same way you would for the ADOTable. Alternatively, you can double-click an ADOConnection component (or use a specific item of its Component Editor, in its shortcut menu) to invoke the connection string editor directly. With the `ConnectionString` set to the proper database, you can disable the login dialog box by setting `LoginPrompt` to False. To use the new connection in the previous example, set the `Connection` property of `ADOTable1` to `ADOConnection1`. You will see `ADOTable1`'s `ConnectionString` property reset because the `Connection` and `ConnectionString` properties are mutually exclusive. One of the benefits of using an ADOConnection is that the connection string is centralized instead of scattered throughout many components. Another, more important, benefit is that all the components that share the ADOConnection share a single connection to the database server. Without your own ADOConnection, each ADO dataset has a separate connection.

Data Link Files

So, an ADOConnection allows you to centralize the definition of a connection string within a form or data module. However, even though this is a worthwhile step forward from scattering the same connection string throughout all ADO datasets, it still suffers from a fundamental flaw: If you use a database engine that defines the database in terms of a filename, then the path to the database file(s) is hard-coded in the EXE. This makes for a fragile application. To overcome this problem, ADO uses Data Link files.

A Data Link file is a connection string in an INI file. For example, Delphi's installation adds to the system the `dbdemos.udl` file, with the following text:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;
  Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb
```

Although you can give a Data Link file any extension, the recommended extension is `.UDL`. You can create a Data Link using any text editor, or you can right-click in Windows Explorer, select New, select Text Document, rename the file with a `.UDL` extension (assuming extensions are displayed in your configuration of Explorer), and then double-click the file to invoke the Microsoft connection string editor.

When you select a file in the Connection editor, the `ConnectionString` property will be set to 'FILE NAME =' followed by the actual filename, as demonstrated by the DataLinkFile example. You can place your Data Link files anywhere on the hard disk, but if you are looking for a common, shared location, then you can use the `DataLinkDir` function in the ADODB Delphi unit. If you haven't altered MDAC's defaults, `DataLinkDir` will return the following:

```
C:\Program Files\Common Files\System\OLE DB\Data Links
```

Dynamic Properties

Imagine that you are responsible for designing a new database middleware architecture. You have to reconcile two opposing goals of a single API for all databases and access to database-specific features. You could take the approach of designing an interface that is the sum of all the features of every database ever created. Each class would have every property and method imaginable, but it would only use the properties and methods it had support for. It doesn't take much discussion to realize that this isn't a good solution. ADO has to solve these apparently mutually exclusive goals, and it does so using *dynamic properties*.

Almost all ADO interfaces and their corresponding dbGo components have a property called `Properties` that is a collection of database-specific properties. These properties can be accessed by their ordinal position, like this:

```
ShowMessage(ADOTable1.Properties[1].Value);
```

But they are more usually accessed by name:

```
ShowMessage(ADOConnection1.Properties['DBMS Name'].Value);
```

Dynamic properties depend on the type of object and also on the OLE DB providers. To give you an idea of their importance, a typical ADO Connection or Recordset has approximately 100 dynamic properties. As you will see throughout this chapter, the answers to many ADO questions lie in dynamic properties.

TIP An important event related to the use of dynamic properties is `OnRecordsetCreate`*, which was introduced in a Delphi 6 update and is available in Delphi 7.* `OnRecordsetCreate` *is called immediately after the recordset has been created, but before it is opened. This event is useful when you're setting dynamic properties that can be set only when the recordset is closed.*

Getting Schema Information

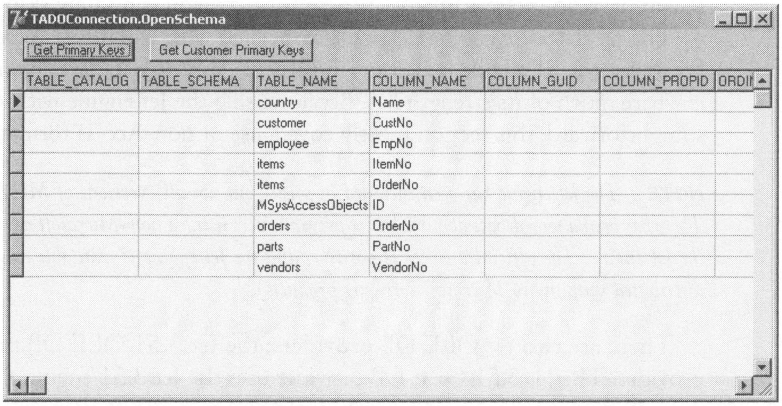
In ADO, you can retrieve schema information using the ADOConnection component's `OpenSchema` method. This method accepts four parameters:

- The kind of data `OpenSchema` should return. It is a `TSchemaInfo` value: a set of 40 values including those for retrieving a list of tables, indexes, columns, views, and stored procedures.
- A filter to place on the data before it is returned. You will see an example of this parameter in a moment.
- A GUID for a provider-specific query. This parameter is used only if the first parameter is siProviderSpecific.
- An ADODataSet into which the data is returned. This parameter illustrates a common theme in ADO: Any method that needs to return more than a small amount of data will return its data as a Recordset, or, in Delphi terms, an ADODataSet.

To use `OpenSchema`, you need an open ADOConnection. The following code (part of the OpenSchema example) retrieves a list of primary keys for every table into an ADODataSet:

```
ADOConnection1.OpenSchema(siPrimaryKeys, EmptyParam, EmptyParam, ADODataSet1);
```

Each field in a primary key has a single row in the result set. So, a table with a composite key of two fields has two rows. The two `EmptyParam` values indicate that these parameters are given empty values and are ignored. The result of this code is shown in Figure 15.3, after resizing the grid with some custom code.



TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	COLUMN_GUID	COLUMN_PROPID	ORDI
		country	Name			
		customer	CustNo			
		employee	EmpNo			
		items	ItemNo			
		items	OrderNo			
		MSysAccessObjects	ID			
		orders	OrderNo			
		parts	PartNo			
		vendors	VendorNo			

FIGURE 15.3
The OpenSchema example retrieves the primary keys of the database tables.

When `EmptyParam` is passed as the second parameter, the result set includes all information of the requested type for the entire database. For many kinds of information, you will want to filter the result set. You can, of course, apply a traditional Delphi filter to the result set using the `Filter` and `Filtered` properties or the `OnFilterRecord` event. However, doing so applies the filter on the client side in this example. Using the second parameter, you can apply a more efficient filter at the source of the schema information. The filter is specified as an array of values. Each element of the array has a specific meaning relevant to the kind of data being returned. For example, the filter array for primary keys has three elements: the catalog (*catalog* is ANSI-speak for the database), the schema, and the table name. This example returns a list of primary keys for the Customer table:

```
var
  Filter: OLEVariant;
begin
  Filter := VarArrayCreate([0, 2], varVariant);
  Filter[2] := 'CUSTOMER';
  ADOConnection1.OpenSchema(
  siPrimaryKeys, Filter, EmptyParam, ADODataSet1);
end;
```

NOTE *You can retrieve the same information using ADOX, and this warrants a brief comparison between* `OpenSchema` *and ADOX. ADOX is an additional ADO technology that allows you to retrieve and update schema information. It is ADO's equivalent to SQL's Data Definition Language (DDL—*`CREATE`*,* `ALTER`*,* `DROP`*) and Data Control Language (DCL—*`GRANT`*,* `REVOKE`*). ADOX is not directly supported in dbGo, but you can import the ADOX type library and use it successfully in Delphi applications. Unfortunately, ADOX is not as universally implemented as* `OpenSchema` *so there are greater gaps. To just retrieve information and not update it,* `OpenSchema` *is usually a better choice.*

Using the Jet Engine

Now that you have some of the MDAC and ADO basics under your belt, let's take a moment out to look at the Jet engine. This engine is of great interest to some and of no interest to others. If you're interested in Access, Paradox, dBase, text, Excel, Lotus 1-2-3, or HTML, then this section is for you. If you have no interest in any of these formats, you can safely skip this section.

The Jet database engine is usually associated with Microsoft Access databases, and this is its forte. However, the Jet engine is also an all-purpose desktop database engine, and this lesser-known attribute is where much of its strength lies. Because using the Jet engine with Access is its default mode and is straightforward, this section mostly covers use of non-Access formats, which are not so obvious.

NOTE *The Jet engine has been included in some (but not all) versions of MDAC. It is not included in version 2.6. There has been a long debate about whether programmers using a non-Microsoft development tool have the right to distribute the Jet engine. The official answer is positive, and the Jet engine is available as a free download (in addition to being distributed with many Microsoft software products).*

There are two Jet OLE DB providers: the Jet 3.51 OLE DB provider and the Jet 4.0 OLE DB provider. The Jet 3.51 OLE DB provider uses the Jet 3.51 engine and supports Access 97 databases only. If you intend to use Access 97 and not Access 2000, then you will get better performance using this OLE DB provider in most situations than using the Jet 4.0 OLE DB provider.

The Jet 4.0 OLE DB provider supports Access 97, Access 2000, and Installable Indexed Sequential Access Method (IISAM) drivers. Installable ISAM drivers are those written specifically for the Jet engine to support access to ISAM formats such as Paradox, dBase, and text, and it is this facility that makes the Jet engine so useful and versatile. The complete list of ISAM drivers installed on your machine depends on what software you have installed. You can find this list by looking in the Registry at

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM Formats
```

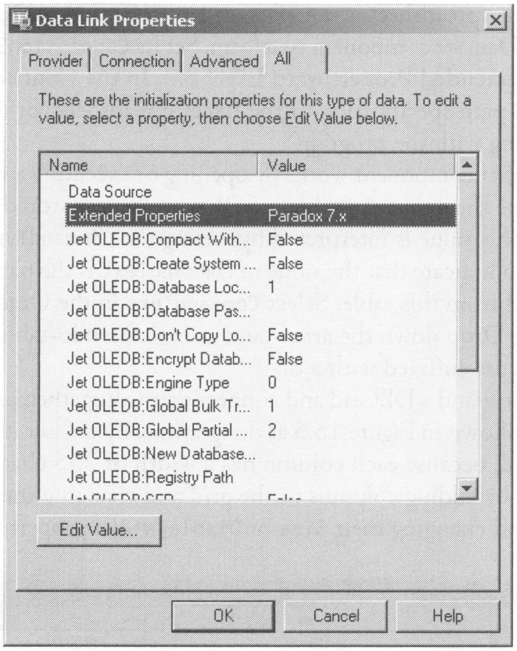
However, the Jet engine includes drivers for Paradox, dBase, Excel, text, and HTML.

Paradox through Jet

The Jet engine expects to be used with Access databases. To use it with any database other than Access, you need to tell it which IISAM driver to use. This is a painless process that involves setting the Extended Properties connection string argument in the connection string editor. Let's work through a quick example.

Add an ADOTable component to a form and invoke the connection string editor. Select the Jet 4.0 OLE DB Provider. Select the All page, locate the Extended Properties property, and double-click it to edit its value.

Enter **Paradox 7.x** in the Property Value, as illustrated in Figure 15.4, and click OK. Now go back to the Connection tab and enter the name of the directory containing the Paradox tables directly, because the Browse button won't help you (it lets you enter a filename, not a folder name). At this point you can select a table in the ADOTable's `TableName` and open it either at design time or at run time. You are now using Paradox through ADO, as demonstrated by the JetParadox example.



These are the initialization properties for this type of data. To edit a value, select a property, then choose Edit Value below.

Name	Value
Data Source	
Extended Properties	Paradox 7.x
Jet OLEDB:Compact With...	False
Jet OLEDB:Create System ...	False
Jet OLEDB:Database Loc...	1
Jet OLEDB:Database Pas...	
Jet OLEDB:Don't Copy Lo...	False
Jet OLEDB:Encrypt Datab...	False
Jet OLEDB:Engine Type	0
Jet OLEDB:Global Bulk Tr...	1
Jet OLEDB:Global Partial ...	2
Jet OLEDB:New Database...	
Jet OLEDB:Registry Path	

FIGURE 15.4
Setting extended properties

I have some bad news for Paradox users: Under certain circumstances, you will need to install the BDE in addition to the Jet engine. Jet 4.0 requires the BDE in order to be able to update Paradox tables, but it doesn't require the BDE just to read them. The same is true for most releases of the Paradox ODBC Driver. Microsoft has received justified criticism about this point and has made a new Paradox IISAM available that does not require the BDE; you can get these updated drivers from Microsoft Technical Support.

NOTE *As you learn more about ADO, you will discover how much it depends on the OLE DB provider and the RDBMS (relational database management system) in question. Although you can use ADO with a local file format, as demonstrated in this and following examples, the general suggestion is to install a local SQL engine whenever possible. Access and MSDE are good choices if you have to use ADO; otherwise you might want to consider InterBase or Firebird as alternatives, as discussed in Chapter 14.*

Excel through Jet

Excel is easily accessed using the Jet OLE DB provider. Once again, you set the Extended Properties property to Excel 8.0. Assume that you have an Excel spreadsheet called `ABCCompany.xls` with a sheet called Employees, and you want to open and read this file using Delphi. With a little knowledge of COM, you can do so by automating Excel. However, the ADO solution is considerably easier to implement and doesn't require Excel to be available on the computer.

TIP *You can also read an Excel file using the XLSReadWrite component (available from* `www.axolot.com`*). It doesn't require Excel to be available on the computer or the time to start it (like OLE Automation techniques do).*

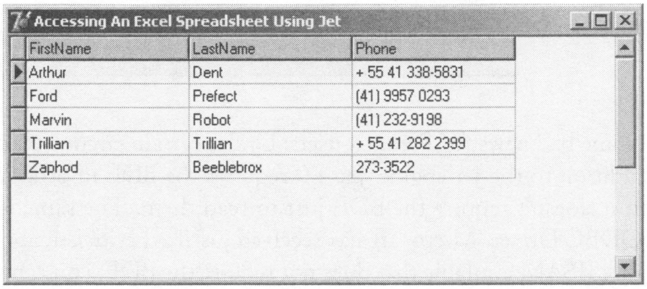
Ensure that your spreadsheet is not open in Excel, because ADO requires exclusive access to the file. Add an ADODataSet component to a form. Set its `ConnectionString` to use the Jet 4.0 OLE DB provider and set Extended Properties to Excel 8.0. In the Connection tab, set the database name to the full file and path specification of the Excel spreadsheet (or use a relative path if you plan to deploy the file along with the program).

The ADODataSet component works by opening or executing a value in its `CommandText` property. This value might be the name of a table, a SQL statement, a stored procedure, or the name of a file. You specify how this value is interpreted by setting the `CommandType` property. Set `CommandType` to cmdTableDirect to indicate that the value in `CommandText` is the name of a table and that all columns should be returned from this table. Select `CommandText` in the Object Inspector, and you will see a drop-down arrow. Drop down the arrow and a single pseudo-table will be displayed: Employees$. (Excel workbooks are suffixed with a $.)

Add a DataSource and a DBGrid and connect them altogether, and you'll obtain the output of the JetExcel example, shown in Figure 15.5 at design time. By default it would be a little difficult to view the data in the grid, because each column has a width of 255 characters. You can change the field display size either by adding columns to the grid and changing their `Width` properties or by adding persistent fields and changing their `Size` or `DisplayWidth` properties.

FIGURE 15.5

ABCCompany.xls in Delphi—a small tribute to Douglas Adams



FirstName	LastName	Phone
Arthur	Dent	+ 55 41 338-5831
Ford	Prefect	(41) 9957 0293
Marvin	Robot	(41) 232-9198
Trillian	Trillian	+ 55 41 282 2399
Zaphod	Beeblebrox	273-3522

Notice that you cannot keep the dataset open at design time and run the program, because the Excel IISAM driver opens the XLS file is in exclusive mode. Close the dataset and add to the program a line of code to open it at startup. When you run the program, you will notice another limitation of this IISAM driver: You can add new rows and edit existing rows, but you cannot delete rows.

Incidentally, you could have used either an ADOTable or an ADOQuery component, instead of the ADODataSet, but you need to be aware of how ADO treats symbols in things like table names and field names. If you use an ADOTable and drop down the list of tables, you will see the Employees$ table as you expect. Unfortunately, if you attempt to open the table, you will receive an error. The same is true for `SELECT * FROM Employees$` in a `TADOQuery`. The problem lies with the dollar sign in the table name. If you use characters such as dollar signs, dots, or, more importantly, spaces in a table name or field name, then you must enclose the name in square brackets (for example, `[Employees$]`).

Text Files through Jet

One of the most useful IISAM drivers that comes with the Jet engine is the Text IISAM. This driver allows you to read and update text files of almost any structured format. We will begin with a simple text file and then cover the variations.

Assume you have a text file called `NightShift.TXT` that contains the following text:

```
CrewPerson ,HomeTown
Neo        ,Cincinnati
Trinity    ,London
Morpheus   ,Milan
```

Add an ADOTable component to a form, set its `ConnectionString` to use the Jet 4.0 OLE DB provider, and set Extended Properties to Text. The Text IISAM provider considers a directory a database, so you need to enter as the database name the directory that contains the `NightShift.TXT` file. In the Object Inspector and drop down the list of tables in the `TableName` property. You will notice that the dot in the filename has been converted to a hash, as in `NightShift#TXT`. Set `Active` to True, add a DataSource and a DBGrid and connect them altogether, and you will see the contents of the text file in a grid.

WARNING *If your computer's settings are such that the decimal separator is a comma instead of a period (so that 1,000.00 is displayed as 1.000,00), then you will need to either change your Regional Settings (Start ➢ Settings ➢ Control Panel ➢ Regional Settings ➢ Numbers) or take advantage of* `SCHEMA.INI`*, described shortly.*

The grid indicates that the widths of the columns are 255 characters. You can change these values just as you did in the JetExcel program by adding persistent fields or columns to the grid and then setting the relevant width property. Alternatively, you can define the structure of the text file more specifically using `SCHEMA.INI`.

In the JetText example, the database folder is determined at run time depending on the folder hosting the program. To modify the connection string at run time, first load it into a string list (after converting the separators) and then use the `Values` property to change only one of the elements of the connection string. This is the code from the example:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  sl.Text := StringReplace (ADOTable1.ConnectionString,
    ';', sLineBreak, [rfReplaceAll]);
  sl.Values ['Data Source'] := ExtractFilePath (Application.ExeName);
  ADOTable1.ConnectionString := StringReplace (sl.Text,
    sLineBreak, ';', [rfReplaceAll]);
  ADOTable1.Open;
  sl.Free;
end;
```

Text files come in all shapes and sizes. Often you do not need to worry about the format of a text file because the Text IISAM takes a peek at the first 25 rows to see whether it can determine the format for itself. It uses this information and additional information in the Registry to decide how to interpret the file and how to behave. If you have a file that doesn't match a regular format the Text IISAM can determine, then you can provide this information using a `SCHEMA.INI` file located in the same directory as the text files to which it refers. This file contains *schema* information, also called metadata, about any or all of the text files in the same directory. Each text file is given its own section, identified by the name of the text file, such as `[NightShift.TXT]`.

Thereafter you can specify the format of the file; the names, types, and sizes of columns; any special character sets to use; and any special column formats (such as date/time or currency). Let's assume that you change your `NightShift.TXT` file to the following format:

```
Neo       |Cincinnati
Trinity   |London
Morpheus  |Milan
```

In this example, the column names are not included in the text file, and the delimiter is a vertical bar. An associated `SCHEMA.INI` file might look something like the following:

```
[NightShift.TXT]
Format=Delimited(|)
ColNameHeader=False
Col1=CrewPerson Char Width 10
Col2=HomeTown Char Width 30
```

Regardless of whether you use a `SCHEMA.INI` file, you will encounter two limitations of the Text IISAM: Rows cannot be deleted, and rows cannot be edited.

Importing and Exporting

The Jet engine is particularly adept at importing and exporting data. The process of exporting data is the same for each export format and consists of executing a `SELECT` statement with a special syntax. Let's begin with an example of exporting data from the Access version of the DBDemos database back to a Paradox table. You will need an active ADOConnection, called `ADOConnection1` in the JetImportExport example, which uses the Jet engine to open the database. The following code exports the Customer table to a Paradox `Customer.db` file:

```
SELECT * INTO Customer IN "C:\tmp" "Paradox 7.x;" FROM CUSTOMER
```

Let's look at the pieces of this `SELECT` statement. The `INTO` clause specifies the new table that will be created by the `SELECT` statement; this table must not already exist. The `IN` clause specifies the database to which the new table is added; in Paradox, this is a directory that already exists. The clause immediately following the database is the name of the IISAM driver to be used to perform the export. *You must include the trailing semicolon at the end of the driver name.* The `FROM` clause is a regular part of any `SELECT` statement. In the sample program, the operation is executed through the ADOConnection component and uses the program's folder instead of a fixed one:

```
ADOConnection1.Execute ('SELECT * INTO Customer IN "' +
  CurrentFolder + '" "Paradox 7.x;" FROM CUSTOMER');
```

All export statements follow these same basic clauses, although IISAM drivers have differing interpretations of what a database is. Here, you export the same data to Excel:

```
ADOConnection1.Execute ('SELECT * INTO Customer IN "' +
  CurrentFolder + 'dbdemos.xls" "Excel 8.0;" FROM CUSTOMER');
```

A new Excel file called `dbdemos.xls` is created in the application's current directory. A workbook called Customer is added, containing all the data from the Customer table in `dbdemos.mdb`.

This last example exports the same data to an HTML file:

```
ADOConnection1.Execute ('SELECT * INTO [Customer.htm] IN "' +
  CurrentFolder + '" "HTML Export;" FROM CUSTOMER');
```

In this case, the database is the directory, as it was for Paradox but not for Excel. The table name must include the `.htm` extension and, therefore, it must be enclosed in square brackets. Notice that the name of the IISAM driver is `HTML Export`, not just `HTML`, because this driver can only be used for exporting to HTML.

The last IISAM driver we'll look at in this investigation of the Jet engine is the partner to HTML Export: HTML Import. Add an ADOTable to a form, set its `ConnectionString` to use the Jet 4.0 OLE DB provider, and set Extended Properties to HTML Import. Set the database name to the name of the HTML file created by the export a few moments ago—that is, `Customer.htm`. Now set the `TableName` property to Customer. Open the table—you have just imported the HTML file. Bear in mind, though, that if you attempt to update the data, you'll receive an error because this driver is intended for import only. Finally, if you create your own HTML files containing tables and want to open these tables using this driver, then remember that the name of the table is the value of the `caption` tag of the HTML `table`.

Working with Cursors

Two properties of ADO datasets have a fundamental impact on your application and are inextricably linked with each other: `CursorLocation` and `CursorType`. If you want to understand your ADO dataset behavior, you must understand these two properties.

Cursor Location

The `CursorLocation` property allows you to specify what is in control of the retrieval and update of your data. You have two choices: client (clUseClient) or server (clUseServer). Your choice affects your dataset's functionality, performance, and scalability.

A client cursor is managed by the ADO Cursor Engine. This engine is an excellent example of an OLE DB service provider: It provides a service to other OLE DB providers. The ADO Cursor Engine manages the data from the client side of the application. All data in the result set is retrieved from the server to the client when the dataset is opened. Thereafter, the data is held in memory, and updates and manipulation are managed by the ADO Cursor Engine. This is similar to using the ClientDataSet component in a dbExpress application. One benefit is that manipulation of the data, after the initial retrieval, is considerably faster. Furthermore, because the manipulation is performed in memory, the ADO Cursor Engine is more versatile than most server-side cursors and offers extra facilities. I'll examine these benefits later, as well as other technologies that depend on client-side cursors (such as disconnected and persistent recordsets).

A server-side cursor is managed by the RDBMS. In a client/server architecture based on a database such as SQL Server, Oracle, or InterBase, this means the cursor is managed physically on the server. In a desktop database such as Access or Paradox, the "server" location is a logical location, because the database is running on the desktop. Server-side cursors are often faster to load than client-side cursors because not all the data is transferred to the client when the dataset is opened. This behavior also makes them more suitable for very large result sets where the client has insufficient memory to hold the entire result set in memory. Often you can determine what kinds of features will be available to you with each cursor location by thinking through how the cursor works. Locking is a good example of how features determine the cursor type; I will discuss locking in more detail later. (To place a lock on a record requires a server-side cursor, because there must be a conversation between the application and the RDBMS.)

Another issue that will affect your choice of cursor location is scalability. Server-side cursors are managed by the RDBMS; in a client/server database, this will be located on the server. As more users use your application, the load on the server increases with each server-side cursor. A greater workload on the server means that the RDBMS becomes a bottleneck more quickly, so the application is less scalable. You can achieve better scalability by using client-side cursors. The initial hit on opening the cursor is often heavier, because all the data is transferred to the client, but the maintenance of the open cursor can be lower. As you can see, many conflicting issues are involved in choosing the correct cursor location for your datasets.

Cursor Type

Your choice of cursor location directly affects your choice of cursor type. To all intents and purposes there are four cursor types, but one value is unused: a cursor type that means *unspecified*. Many values in ADO signify an unspecified value, and I will cover them all here and explain why you won't have much to do with them. They exist in Delphi because they exist in ADO. ADO was primarily designed for Visual Basic and C programmers. In these languages, you use objects directly without the assistance dbGo provides. As such, you can create and open *recordsets*, as they are called in ADO-speak, without having to specify every value for every property. The properties for which a value has not been specified have an unspecified value. However, in dbGo you use components. These components have constructors, and these constructors initialize the properties of the components. So, from the moment you create a dbGo component, it will usually have a value for every property. As a consequence, you have little need for the unspecified values in many enumerated types.

Cursor types affect how your data is read and updated. As I mentioned, there are four choices: forward-only, static, keyset, and dynamic. Before we get too involved in all the permutations of cursor locations and cursor types, you should be aware that there is only one cursor type available for client-side cursors: the static cursor. All other cursor types are available only to server-side cursors. I'll return to the subject of cursor type availability after we have looked at the various cursor types, in increasing order of expense:

Forward-Only Cursor The forward-only cursor is the least expensive cursor type, and therefore the type with the best performance. As the name implies, the forward-only cursor lets you navigate forward. The cursor reads the number of records specified by `CacheSize` (default of 1); each time it runs out of records, it reads another `CacheSize` set. Any attempt to navigate backward through the result set beyond the number of records in the cache results in an error. This behavior is similar to that of a dbExpress dataset. A forward-only cursor is not suitable for use in the user interface where the user can control the direction through the result set. However, it is eminently suitable for batch operations, reports and stateless Web applications, because these situations start at the top of the result set and work progressively toward the end, and then the result set is closed.

Static Cursor A static cursor works by reading the complete result set and providing a window of `CacheSize` records into the result set. Because the complete result set has been retrieved by the server, you can navigate both forward and backward through the result set. However, in exchange for this facility, the data is static—updates, insertions, and deletions made by other users cannot be seen, because the cursor's data has already been read.

Keyset Cursor A keyset cursor is best understood by breaking *keyset* into the two words *key* and *set*. *Key*, in this context, refers to an identifier for each row. Often this will be a primary key. A keyset cursor, therefore, is a set of keys. When the result set is opened, the complete list of keys for the result set is read. If, for example, the dataset was a query like `SELECT * FROM CUSTOMER`, then the list of keys would be built from `SELECT CUSTID FROM CUSTOMER`. This set of keys is held until the cursor is closed. When the application requests data, the OLE DB provider reads the rows using the keys in the set of keys. Consequently, the data is always up to date. If another user changes a row in the result set, then the changes will be seen when the data is reread. However, the set of keys is static; it is read only when the result set is first opened. If another user adds new records, these additions will not be seen. Deleted records become inaccessible, and changes to primary keys (you don't let your users change primary keys, do you?) are also inaccessible.

Dynamic Cursor The most expensive cursor type, a dynamic cursor is almost identical to a keyset cursor. The sole difference is that the set of keys is reread when the application requests data that is not in the cache. Because the default for `ADODataSet.CacheSize` is 1, such requests occur frequently. You can imagine the additional load this behavior places on the DBMS and the network, and why this is the most expensive cursor. However, the result set can see and respond to additions and deletions made by other users.

Ask and Ye Shall Not Receive

Now that you know about cursor locations and cursor types, a word of warning: Not all combinations of cursor location and cursor type are possible. Usually, this limitation is imposed by the RDBMS and/or the OLE DB provider as a result of the functionality and architecture of the database. For example, client cursors always force the cursor type to static. You can see behavior this for yourself. Add an ADODataSet component to a form, set its `ConnectionString` to any database, and set the `ClientLocation` property to clUseCursor and the `CursorType` property to ctDynamic. Now set `Active` to True and keep your eye on the `CursorType`; it changes to ctStatic. You learn an important lesson from this example: What you ask for is not necessarily what you get. Always check your properties after opening a dataset to see the actual effect of your requests.

Each OLE DB provider will make different changes according to different requests and circumstances, but here are a few examples to give you an idea of what to expect:

- The Jet 4.0 OLE DB provider changes most cursor types to keyset.
- The SQL Server OLE DB provider often changes keyset and static to dynamic.
- The Oracle OLE DB provider changes all cursor types to forward-only.
- The ODBC OLE DB provider makes various changes according to the ODBC driver in use.

No Record Count

ADO datasets sometimes return –1 for their `RecordCount`. A forward-only cursor cannot know how many records are in the result set until it reaches the end, so it returns –1 for the `RecordCount`. A static cursor always knows how many records are in the result set, because it reads the entire set when it is opened, so it returns the number of records in its result set. A keyset cursor also knows how many records are in the result set, because it has to retrieve a fixed set of keys when the result set is opened, so it also returns a useful value for `RecordCount`. A dynamic cursor does not reliably know how many records are in the result set, because it is regularly rereading the set of keys, so it returns –1. You can avoid using `RecordCount` altogether and execute `SELECT COUNT(*) FROM tablename`, but the result will be an accurate reflection of the number of records in the database table—which is not necessarily the same as the number of records in the dataset.

Client Indexes

One of the many benefits of client-side cursors is the ability to create local, or *client,* indexes. Assuming you have an ADO client-side dataset for the DBDemos Customer table, which has a grid attached to it, set the dataset's `IndexFieldNames` property to CompanyName. The grid will immediately show that the data is in CompanyName order. There is an important point here: In order to index the data, ADO did not have to reread the data from its source. The index was created from the data in memory. So, not only is the index created as quickly as possible, but the network and the DBMS are not overloaded by transferring the same data over and over in different orders.

The `IndexFieldNames` property has more potential. Set it to `Country;CompanyName` and you will see the data ordered first by country and then, within country, by company name. Now set `IndexFieldNames` to `CompanyName DESC`. Be sure to write *DESC* in capitals (and not *desc* or *Desc*). The data is now sorted in descending order.

This simple but powerful feature allows you to solve one of the great bugbears of database developers. Users ask the inevitable, and quite reasonable, question, "Can I click the columns of the grid to sort my data?" Answers like replacing grids with non–data-aware controls such as ListView that have the sorting built into the control or like trapping the DBGrid's `OnTitleClick` event and reissuing the SQL `SELECT` statement after including an appropriate `ORDER BY` clause are far from satisfactory.

If you have the data cached on the client side (as you've also seen in the use of the ClientDataSet component), you can use a client index computed in memory. Add the following `OnTitleClick` event to the grid (the code is available in the ClientIndexes example):

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  if ADODataSet1.IndexFieldNames = Column.Field.FieldName then
    ADODataSet1.IndexFieldNames := Column.Field.FieldName + ' DESC'
  else
    ADODataSet1.IndexFieldNames := Column.Field.FieldName
end;
```

This simple event checks to see whether the current index is built on the same field as the column. If it is, then a new index is built on the column, but in descending order. If not, then a new index is built on the column. When the user clicks the column for the first time, it is sorted in ascending order; when it is clicked for the second time, it is sorted in descending order. You could extend this functionality to allow the user to Ctrl+click several column titles to build up more complicated indexes.

NOTE *All of this can be achieved using ClientDataSet, but that solution is not as elegant because ClientDataSet does not support the DESC keyword so you have to create an index collection item to obtain a descending index, something that requires more code. Moreover when changing an ascending index to a descending version, the ClientDataSet will rebuild the index, performing an unnecessary and possibly slow operation.*

Cloning

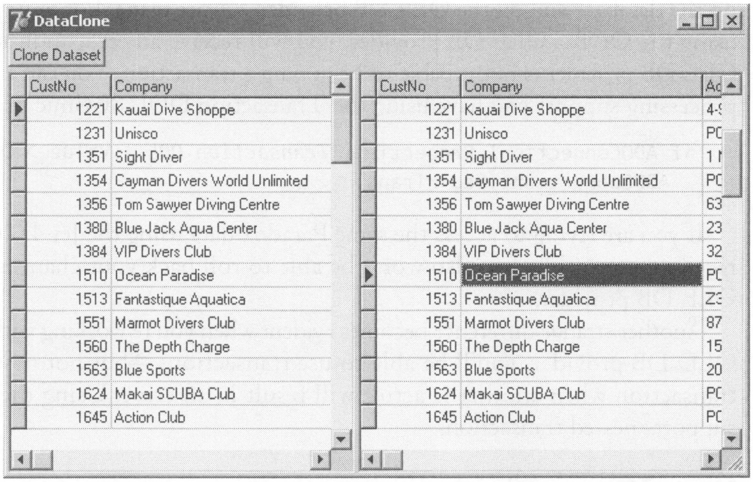
ADO is crammed with features. You can argue that *feature-rich* can translate into *footprint-rich*, but it also translates into more powerful and reliable applications. One such powerful feature is cloning. A cloned recordset is a new recordset that has all the same properties as the original from which it is cloned. First I'll explain how you can create and use a clone, and then I'll explain why clones are so useful.

NOTE *The ClientDataSet also supports cloning; this feature was not discussed in Chapter 13, "Delphi's Database Architecture."*

You can clone a recordset (or, in dbGo-speak, a dataset) using the `Clone` method. You can clone any ADO dataset, but you will use ADOTable in this example. The DataClone example (see Figure 15.6) uses two ADOTable components, one hooked to database data and the other empty. Both datasets are hooked to a DataSource and grid. A single line of code, executed when the user clicks a button, clones the dataset:

```
ADOTable2.Clone(ADOTable1);
```

FIGURE 15.6 The form of the DataClone example, with two copies of a dataset (the original and a clone)



	CustNo	Company
▶	1221	Kauai Dive Shoppe
	1231	Unisco
	1351	Sight Diver
	1354	Cayman Divers World Unlimited
	1356	Tom Sawyer Diving Centre
	1380	Blue Jack Aqua Center
	1384	VIP Divers Club
	1510	Ocean Paradise
	1513	Fantastique Aquatica
	1551	Marmot Divers Club
	1560	The Depth Charge
	1563	Blue Sports
	1624	Makai SCUBA Club
	1645	Action Club

	CustNo	Company	Ad
	1221	Kauai Dive Shoppe	4-
	1231	Unisco	P
	1351	Sight Diver	1
	1354	Cayman Divers World Unlimited	P
	1356	Tom Sawyer Diving Centre	63
	1380	Blue Jack Aqua Center	23
	1384	VIP Divers Club	32
▶	1510	Ocean Paradise	P
	1513	Fantastique Aquatica	Z
	1551	Marmot Divers Club	87
	1560	The Depth Charge	15
	1563	Blue Sports	20
	1624	Makai SCUBA Club	P
	1645	Action Club	P

This line clones `ADOTable1` and assigns the clone to `ADOTable2`. In the program, you'll see a second view of the data. The two datasets have their own record pointers and other status information, so the clone does not interfere with its original copy. This behavior makes clones ideal for operating on a dataset without affecting the original data. Another interesting feature is that you can have multiple different active records, one for each of the clones—functionality you cannot achieve in Delphi with a single dataset.

TIP *A recordset must support bookmarks in order to be cloned, so forward-only and dynamic cursors cannot be cloned. You can determine whether a recordset supports bookmarks using the* `Supports` *method (for example,* `ADOTable1.Supports([coBookMark])`*). One of the useful side effects of clones is that the bookmarks created by one clone are usable by all other clones.*

Transaction Processing

As we saw in section "Using Transactions" of Chapter 14, transaction processing allows developers to group individual updates to a database into a single logical unit of work.

ADO's transaction processing support is controlled with the ADOConnection component, using the `BeginTrans`, `CommitTrans`, and `RollbackTrans` methods, which have effects similar to those of the corresponding dbExpress and BDE methods. To investigate ADO transaction processing support, you will build a test program called TransProcessing. The program has an ADOConnection component with the `ConnectionString` set to the Jet 4.0 OLE DB provider and the `dbdemos.mdb` file. It has an ADOTable component hooked to the Customer table and a DataSource and DBGrid for displaying the data. Finally, it has three buttons to execute the following commands:

```
ADOConnection1.BeginTrans;
ADOConnection1.CommitTrans;
ADOConnection1.RollbackTrans;
```

With this program, you can make changes to the database table and then roll them back, and they will be rolled back as expected. I emphasize this point because transaction support varies depending on the database and on the OLE DB provider you are using. For example, if you connect to Paradox using the ODBC OLE DB provider, you will receive an error indicating that the database or the OLE DB provider is not capable of beginning a transaction. You can find out the level of transaction processing support you have using the Transaction DDL dynamic property of the connection:

```
if ADOConnection1.Properties['Transaction DDL'].Value > DBPROPVAL_TC_NONE then
  ADOConnection1.BeginTrans;
```

If you are trying to access the same Paradox data using the Jet 4.0 OLE DB provider, you won't receive an error but you also won't be able to roll back your changes, due to limitations of the OLE DB provider.

Another strange difference becomes evident when you're working with Access: If you use the ODBC OLD DB provider, you'll be able to use transactions—but not nested transactions. Opening a transaction when another is active will result in an error. Using the Jet engine, however, Access supports nested transactions.

Nested Transactions

Using the TransProcessing program, you can try this test:

1. Begin a transaction.
2. Change the `ContactName` of the Around The Horn record from Thomas Hardy to Dick Solomon.
3. Begin a nested transaction.
4. Change the `ContactName` of the Bottom-Dollar Markets record from Elizabeth Lincoln to Sally Solomon.
5. Roll back the inner transaction.
6. Commit the outermost transaction.

The net effect is that only the change to the Around The Horn record is permanent. If, however, the inner transaction had been committed and the outer transaction rolled back, then the net effect would have been that *none* of the changes were permanent (even the changes in the inner transaction). This is as you would expect, with the only limit being that Access supports five levels of nested transactions.

ODBC does not support nested transactions, the Jet OLE DB provider supports up to five levels of nested transactions, and the SQL Server OLE DB provider doesn't support nesting at all. You might get a different result depending on the version of SQL server or the driver, but the documentation and my experiments with the servers seem to indicate that this is the case. Apparently only the outermost transaction decides whether all the work is committed or rolled back.

ADOConnection Attributes

There is another issue you should consider if you intend to use nested transactions. The ADOConnection component has an `Attributes` property that determines how the connection should behave when a transaction is committed or rolled back. It is a set of `TXActAttributes` that, by default, is empty. `TXActAttributes` contains only two values: xaCommitRetaining and xaAbortRetaining (this value is often mistakenly written as xaRollbackRetaining—a more logical name for it). When xaCommitRetaining is included in `Attributes` and a transaction is committed, a new transaction is automatically started. When xaAbortRetaining is included in `Attributes` and a transaction is rolled back, a new transaction is automatically started. Thus if you include these values in `Attributes`, a transaction will always be in progress, because when you end one transaction another will always be started.

Most programmers prefer to be in greater control of their transactions and not to allow them to be automatically started, so these values are not commonly used. However, they have a special relevance to nested transactions. If you nest a transaction and set `Attributes` to `[xaCommitRetaining, xaAbortRetaining]`, then the outermost transaction can never be ended. Consider this sequence of events:

1. An outer transaction is started.
2. An inner transaction is started.
3. The inner transaction is committed or rolled back.
4. A new inner transaction is automatically started as a consequence of the `Attributes` property.

The outermost transaction can never be ended, because a new inner transaction will be started when one ends. The conclusion is that the use of the `Attributes` property and the use of nested transactions should be considered mutually exclusive.

Lock Types

ADO supports four different approaches to locking your data for update: ltReadOnly, ltPessimistic, ltOptimistic, and ltBatchOptimistic (there is also an ltUnspecified option, but for the reasons mentioned earlier, we will ignore unspecified values). The four approaches are made available through the dataset's `LockType` property. In this section I will provide an overview of the four approaches, and in subsequent sections we will take a closer look at them.

The ltReadOnly value specifies that data is read-only and cannot be updated. As such, there is effectively no locking control required, because the data cannot be updated.

The ltPessimistic and ltOptimistic values offer the same pessimistic and optimistic locking control as the BDE. One important benefit that ADO offers over the BDE in this respect is that the choice of locking control remains yours. If you use the BDE, the decision to use pessimistic or optimistic locking is made for you by your BDE driver. If you use a desktop database such as dBase or Paradox, then the BDE driver uses pessimistic locking; if you use a client/server database such as InterBase, SQL Server, or Oracle, the BDE driver uses optimistic locking.

PESSIMISTIC LOCKING

The words *pessimistic* and *optimistic* in this context refer to the developer's expectation of conflict between user updates. Pessimistic locking assumes that there is a high probability that users will attempt to update the same records at the same time and that a conflict is likely. In order to prevent such a conflict, the record is locked when the edit begins. The record lock is maintained until the update is completed or cancelled. A second user who attempts to edit the same record at the same time will fail in their attempt to place their record lock and will receive a "Could not update; currently locked" exception.

This approach to locking will be familiar to developers who have worked with desktop databases such as dBase and Paradox. The benefit is that the user knows that if they can begin editing a record, then they will succeed in saving their update. The disadvantage of pessimistic locking is that the user is in control of when the lock is placed and when it is removed. If the user is skilled with the application, then this lock could be as short as a couple of seconds. However, in database terms, a couple of seconds is an eternity. On the other hand, the user might begin an edit and go to lunch, and the record would be locked until the user returns. As a consequence, most proponents of pessimistic locking guard against this eventuality by using a Timer or other such device to time out locks after a certain amount of keyboard and mouse inactivity.

Another problem with pessimistic locking is that it requires a server-side cursor. Earlier we looked at cursor locations and saw that they have an impact on the availability of the different cursor types. Now you can see that cursor locations also have an impact on locking types. Later in this chapter, we will discuss more benefits of client-side cursors; if you choose to take advantage of these benefits, then you'll be unable to use pessimistic locking.

Pessimistic locking is an area of dbGo that changed in Delphi 6 (compared to Delphi 5). This section describes the way pessimistic locking works in versions 6 and 7. To highlight how it works, I've built the PessimisticLocking example. It is similar to other examples in this chapter, but the `CursorLocation` property is set to clUseServer and the `LockType` property is set to ltPessimistic. To use it, run two copies from Windows Explorer and attempt to edit the same record in both running instances of the program: You will fail because the record is locked by another user.

Updating the Data

One of the reasons people use the ClientDataSet component (or turn to cached updates in the BDE) is to make a SQL join updatable. Consider the following SQL equi-join:

```
SELECT * FROM Orders, Customer
WHERE Customer.CustNo=Orders.CustNo
```

This statement provides a list of orders and the customers that placed those orders. The BDE considers any SQL join to be read-only because inserting, updating, and deleting rows in a join is ambiguous. For example, should the insert of a row into the previous join result in a new product and also a new supplier, or just a new product? The ClientDataSet/Provider architecture allows you to specify a primary update table (and advanced features actually not covered in the book) and also customize the updates' SQL, as we partially saw in Chapter 14 and we'll further explore in Chapter 16, "Multitier DataSnap Applications."

ADO supports an equivalent to cached updates called *batch updates,* which are similar to the BDE approach. In the next section we will take a closer look at ADO's batch updates, what they can offer you, and why they are so important. However, in this section you won't need them to solve the problem of updating a join, because in ADO, joins are naturally updatable.

For example, the JoinData example is built around an ADODataset component that uses the previous SQL join. If you run it, you can edit one of the fields and save the changes (by moving off the record). No error occurs, because the update has been applied successfully. ADO, compared to the BDE, has taken a more practical approach to the problem. In an ADO join, each field object knows which underlying table it belongs to. If you update a field in the Products table and post the change, then a SQL `UPDATE` statement is generated to update the field in the Products table. If you change a field in the Products table and a field in the Suppliers table, then two SQL `UPDATE` statements are generated, one for each table.

The insertion of a row into a join follows a similar behavior. If you insert a row and enter values for the Products table only, then a SQL `INSERT` statement is generated for the Products table. If you enter values for both tables, two SQL `INSERT` statements are generated, one for each table. The order in which the statements are executed is important, because the new product might relate to the new supplier, so the new supplier is inserted first.

The biggest problem with the ADO solution can be seen when a row in a join is deleted. The deletion attempt will appear to fail. The exact message you see depends on the version of ADO you are using and the database, but it will be along the lines that you cannot delete the row because other records relate to it. The error message can be confusing. In the current scenario, the error message implies that a product cannot be deleted because there are records that relate to the product, but the error occurs whether the product has any related records or not. The explanation can be found by following the same logic for deletions as for insertions. Two SQL `DELETE` statements are generated: one for the Suppliers table and then another for the Products table. Contrary to appearances, the `DELETE` statement for the Products table succeeds. It is the `DELETE` statement for the Suppliers table that fails, because the supplier cannot be deleted while it still has dependent records.

TIP *If you are curious about the SQL statements that are generated, and you use SQL Server, you can see these statements using SQL Server Profiler.*

Even if you understand how this process works, it's helpful to look at this problem through the users' eyes. From their point of view, when users delete a row in the grid, I would wager that 99 percent of them intend to delete just the product—not both the product and the supplier. Fortunately, you can achieve this result using another dynamic property—in this case, the Unique Table dynamic property. You can specify that deletes refer to just the Products table and not to Suppliers using the following line of code:

```
ADOQuery1.Properties['Unique Table'].Value := 'Products';
```

This value cannot be assigned at design time, so the next best alternative is to place this line in the form's `OnCreate` event.

Batch Updates

When you use batch updates, any changes you make to your records can be made in memory; later, the entire "batch" of changes can be submitted as one operation. This approach offers some performance benefits, but there are more practical reasons why this technology is a necessity: The user might not be connected to the database at the time they make their updates. This would be the case in a briefcase application (which we will return to in the section "The Briefcase Model"), but it can also be the case in web applications that use another ADO technology, Remote Data Services (RDS).

You can enable batch updates in any ADO dataset by setting `LockType` to ltBatchOptimistic before the dataset is opened. In addition, you will need to set the `CursorLocation` to clUseClient, because batch updates are managed by ADO's cursor engine. Hereafter, changes are all made to a *delta* (a list of changes). The dataset looks to all intents and purposes as if the data has changed, but the changes have only been made in memory; they have not been applied to the database. To make the changes permanent, use `UpdateBatch` (equivalent to cached updates' `ApplyUpdates`):

```
ADODataSet1.UpdateBatch;
```

To reject the entire batch of updates, use either `CancelBatch` or `CancelUpdates`. There are many similarities in method and property names between ADO batch updates, BDE cached updates, and ClientDataSet. `UpdateStatus`, for example, can be used exactly the same way as for cached updates to identify records according to whether they have been inserted, updated, deleted, or unmodified. This approach is particularly useful for highlighting records in different colors in a grid or showing their status on a status bar. Some differences between the syntaxes are slight, such as changing `RevertRecord` to `CancelBatch(arCurrent)`; others require more effort.

One useful cached update feature that is not present in ADO batch updates is the dataset's `UpdatesPending` property. This property is true if changes have been made but not yet applied. It's particularly useful in a form's `OnCloseQuery` event:

```
procedure TForm1.FormCloseQuery(
  Sender: TObject; var CanClose: Boolean);
begin
  CanClose := True;
  if ADODataSet1.UpdatesPending then
    CanClose := (MessageDlg('Updates are still pending' #13 +
      'Close anyway?', mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

However, with a little knowledge and ingenuity you can implement a suitable `ADOUpdatesPending` function. The necessary knowledge is that ADO datasets have a property called `FilterGroup`, which is a kind of filter. Unlike a dataset's `Filter` property, which filters the data based on a comparison of the data against a condition, `FilterGroup` filters based on the status of the record. One such status is fgPendingRecords, which includes all records that have been modified but not yet applied. So, to allow the user to look through all the changes they have made so far, you need only execute two lines:

```
ADODataSet1.FilterGroup := fgPendingRecords;
ADODataSet1.Filtered := True;
```

Naturally, the result set will now include the records that have been deleted. The effect you will see is that the fields are left blank, which is not very helpful because you don't know which record has been deleted. (This was not the behavior in the first version of ADOExpress, which displayed the field values of deleted records.)

The ingenuity you need in order to solve the `UpdatesPending` problem involves clones, discussed earlier. The `ADOUpdatesPending` function will set the `FilterGroup` to restrict the dataset to only those changes that have not yet been applied. All you need to do is see whether there are any records in the dataset once the `FilterGroup` has been applied. If there are, then some updates are pending. However, if you do this with the actual dataset, then the setting of the `FilterGroup` will move the record pointer, and the user interface will be updated. The best solution is to use a clone:

```
function ADOUpdatesPending(ADODataSet: TCustomADODataSet): boolean;
var
  Clone: TADODataSet;
begin
  Clone := TADODataSet.Create(nil);
  try
    Clone.Clone(ADODataSet);
    Clone.FilterGroup := fgPendingRecords;
    Clone.Filtered := True;
    Result := not (Clone.BOF and Clone.EOF);
    Clone.Close;
  finally
    Clone.Free;
  end;
end;
```

In this function, you clone the original dataset, set the `FilterGroup`, and check to see whether the dataset is at both beginning of the file and also the end of the file. If it is, then no records are pending.

Optimistic Locking

Earlier we looked at the `LockType` property and saw how pessimistic locking works. In this section, we'll look at optimistic locking, not only because it is the preferred locking type for medium- to high-throughput transactions, but also because it is the locking scheme employed by batch updates.

Optimistic locking assumes there is a low probability that users will attempt to update the same records at the same time and that a conflict is unlikely. As such, the attitude is that all users can edit any record at any time, and you deal with the consequences of conflicts between different users' updates to the same records when the changes are saved. Thus, conflicts are considered an exception to the rule. This means there are no controls to prevent two users from editing the same record at the same time. The first user to save their changes will succeed; the second user's attempt to update the same record might fail. This behavior is essential for briefcase applications (discussed later in the chapter) and web applications, where there is no permanent connection to the database and, therefore, no way to implement pessimistic locking. In contrast with pessimistic locking, optimistic locking has the additional considerable benefit that resources are consumed only momentarily; therefore, the average resource usage is much lower, making the database more scalable.

Let's consider an example. Assume you have an ADODataSet connected to the Customer table of the `dbdemos.mdb` database, with `LockType` set to ltBatchOptimistic, and the contents are displayed in a grid. Assume that you also have a button to call `UpdateBatch`. Run the program twice (it is the BatchUpdates example if you don't want to rebuild it) and begin editing a record in the first copy of the program. Although for the sake of simplicity I'll demonstrate a conflict using just a single machine, the scenario and subsequent events are unchanged when using multiple machines:

1. Choose the Bottom-Dollar Markets company in Canada and change the name to Bottom-Franc Markets.
2. Save the change, move off the record to post it, and click the button to update the batch.
3. In the second copy of the program, locate the same record and change the company name to Bottom-Pound Markets.
4. Move off the record and click the button to update the batch. It will fail.

As with many ADO error messages, the exact message you receive will depend not only on the version of ADO you are using but also on how closely you followed the example. In ADO 2.6, the error message is "Row cannot be located for updating. Some values may have been changed since it was last read." This is the nature of optimistic locking. The update to the record is performed by executing the following SQL statement:

```
UPDATE CUSTOMER SET CompanyName="Bottom-Pound Markets"
WHERE CustomerID="BOTTM" AND CompanyName="Bottom-Dollar Markets"
```

The number of records affected by this `UPDATE` statement is expected to be one, because it locates the original record using the primary key and the contents of the CompanyName field as it was when the record was first read. In this example, however, the number of records affected by the `UPDATE` statement is zero. This result can occur only if the record has been deleted, the record's primary key has changed, or the field that you are changing was changed by someone else. Hence, the update fails.

If the "second user" had changed the ContactName field and not the CompanyName field, then the `UPDATE` statement would have looked like this:

```
UPDATE CUSTOMER SET ContactName="Liz Lincoln"
WHERE CustomerID="BOTTM" AND ContactName="Elizabeth Lincoln"
```

In the example scenario, this statement would have succeeded because the other user didn't change the primary key or the contact name. This behavior is similar to the BDE with the *update where changed* update mode. The `UpdateMode` property of the BDE in ADO is replaced by the Update Criteria dynamic property of a dataset. The following list shows the possible values that can be assigned to this dynamic property:

Constant	Locate Records By
adCriteriaKey	Primary key columns only
adCriteriaAllCols	All columns
adCriteriaUpdCols	Primary key columns and changed columns only
adCriteriaTimeStamp	Primary key columns and a timestamp column only

Don't fall into the trap of thinking that one of these settings is better than another for your whole application. In practice, your choice of setting will be influenced by the contents of each table. Say that the Customer table has just CustomerID, Name, and City fields. In this case, the update of any one of these fields is logically not mutually exclusive with the update of any of the other fields, so a good choice for this table would be adCriteriaUpdCols (the default). If, however, the Customer table included a PostalCode field, then the update of a PostalCode field would be mutually exclusive with the update of the City field by another user (because if the city changes, then so should the postal code, and possibly vice versa). In this case, you could argue that adCriteriaAllCols would be a safer solution.

Another issue to be aware of is how ADO deals with errors during the update of multiple records. Using the BDE's cached updates and ClientDataSet, you can use the `OnUpdateError` event to handle each update error as the error occurs and resolve the problem before moving on to the next record. In ADO, you cannot establish such a dialog. You can monitor the progress and success or failure of the updating of the batch using the dataset's `OnWillChangeRecord` and `OnRecordChangeComplete`, but you cannot revise the record and resubmit it during this process as you can with the BDE and ClientDataSet. There's more: If an error occurs during the update process, the updating does not stop. It continues to the end, until all updates have been applied or have failed. This process can produce an unhelpful and incorrect error message. If more than one record cannot be updated, or the single record that failed is not the last record to be applied, then the error message in ADO 2.6 is "Multiple-step OLE DB operation generated errors. Check each OLE DB status value, if available. No work was done." The last sentence is the problem; it states that "No work was done," but this is incorrect. It is true that no work was done on the record that failed, but other records were successfully applied, and their updates stand.

Resolving Update Conflicts

As a consequence of the nature of applying updates, the approach that you need to take to update the batch is to update the batch, let the individual records fail, and then deal with the failed records once the process is over. You can determine which records have failed by setting the dataset's `FilterGroup` to fgConflictingRecords:

```
ADODataSet1.FilterGroup := fgConflictingRecords;
ADODataSet1.Filtered := True;
```

For each failed record, you can inform the user of three critical pieces of information about each field using the following `TField` properties:

Property	Description
`NewValue`	The value this user changed it to
`CurValue`	The new value from the database
`OldValue`	The value when first read from the database

Users of the ClientDataSet component will be aware of the handy `ReconcileErrorForm` dialog, which wraps up the process of showing the user the old and new records and allows them to specify what action to take. Unfortunately, there is no ADO equivalent to this form, and `TReconcileErrorForm` has been written with ClientDataSet so much in mind that it is difficult to convert it for use with ADO datasets.

I'll point out one last gotcha about using these `TField` properties: They are taken straight from the underlying ADO Field objects to which they refer. This means, as is common in ADO, that you are at the mercy of your chosen OLE DB provider to support the features you hope to use. All is well for most providers, but the Jet OLE DB provider returns the same value for `CurValue` as it does for `OldValue`. In other words, if you use Jet, you cannot determine the value to which the other user changed the field unless you resort to your own measures. Using the OLEDB provider for SQL Server, however, you can access the `CurValue` only after calling the `Resync` method of the dataset with the `AffectRecords` parameter set to adAffectGroup and `ResyncValues` set to adResyncUnderlyingValues, as in the following code:

```
adoCustomers.FilterGroup := fgConflictingRecords;
adoCustomers.Filtered := true;
adoCustomers.Recordset.Resync(adAffectGroup, adResyncUnderlyingValues);
```

Disconnected Recordsets

This knowledge of batch updates allows you to take advantage of the next ADO feature: disconnected recordsets. A *disconnected recordset* is a recordset that has been disconnected from its connection. This feature is impressive because the user cannot tell the difference between a regular recordset and a disconnected one; their feature sets and behavior are almost identical. To disconnect a recordset from its connection, you must set the `CursorLocation` to clUseClient and the `LockType` to ltBatchOptimistic. You then tell the dataset that it no longer has a connection:

```
ADODataSet1.Connection := nil;
```

Hereafter, the recordset will continue to contain the same data, support the same navigational features, and allow records to be added, edited, and deleted. The only relevant difference is that you cannot update the batch because you need to be connected to the server to update the server. You can reconnect the connection (and use `UpdateBatch`) as follows:

```
ADODataSet1.Connection := ADOConnection1;
```

This feature is also available to the BDE and other database technologies by switching over to ClientDataSets, but the beauty of the ADO solution is that you can build your entire application using dbGo dataset components and be unaware of disconnected recordsets. When you discover this feature and want to take advantage of it, you can continue to use the same components you always used.

You might want to disconnect your recordsets for two reasons:

- To keep the total number of connections lower
- To create a briefcase application

I'll discuss keeping down the number of connections in this section and return to briefcase applications later.

Most regular client/server business applications open tables and maintain a permanent connection to their database while the table is open. However, there are usually only two reasons to be connected to the database: to retrieve data and to update data. Suppose you change your regular client/server application so that after the table is opened and the data is retrieved, the dataset is disconnected from the connection and the connection is dropped; your user will be none the wiser, and the application will not need to maintain an open database connection. The following code shows the two steps:

```
ADODataSet1.Connection := nil;
ADOConnection1.Connected := False;
```

The only other point at which a connection is required is when the batch of updates needs to be applied. The update code looks like this:

```
ADOConnection1.Connected := True;
ADODataSet1.Connection := ADOConnection1;
try
  ADODataSet1.UpdateBatch;
finally
  ADODataSet1.Connection := nil;
  ADOConnection1.Connected := False;
end;
```

If you followed this approach throughout the application, the average number of open connections at any one time would be minimal—the connections would be open only for the brief time they were required. The consequence of this change is scalability; the application can cope with significantly more simultaneous users than an application that maintains an open connection. The downside is that reopening the connection can be a lengthy process on some (but not all) database engines, so the application will be slower to update the batch.

Connection Pooling

All this talk about dropping and reopening connections brings us to the subject of connection pooling. Connection pooling—not to be confused with session pooling—allows connections to a database to be reused once you have finished with them. This process happens automatically; if your OLE DB provider supports it and it is enabled, no action is necessary for you to take advantage of connection pooling.

There is a single reason to pool your connections: performance. The problem with database connections is that it can take time to establish a connection. In a desktop database such as Access, this time is typically brief. However, in a client/server database such as Oracle used on a network, this time could be measured in seconds. It makes sense to promote the reuse of such an expensive (in performance terms) resource.

With ADO connection pooling enabled, ADO Connection objects are placed in a pool when the application "destroys" them. Subsequent attempts to create an ADO connection will automatically search the connection pool for a connection with the same connection string. If a suitable connection is found, it is reused; otherwise, a new connection is created. The connections themselves stay in the pool until they are reused, the application closes, or they time out. By default, connections will time out after 60 seconds, but from MDAC 2.5 onward you can set this time-out period using the `HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>\SPTimeout` Registry key. The connection pooling process occurs seamlessly, without the intervention or knowledge of the developer. This process is similar to the BDE database pooling under Microsoft Transaction Server (MTS) and COM+, with the important exception that ADO performs its own connection pooling without the aid of MTS or COM+.

By default, connection pooling is enabled on all MDAC OLE DB providers for relational databases (including SQL Server and Oracle), with the notable exception of the Jet OLE DB provider. If you use ODBC, you should choose between ODBC connection pooling and ADO connection pooling, but you should not use both. From MDAC 2.1 on, ADO connection pooling is enabled and ODBC is disabled.

NOTE *Connection pooling does not occur on Windows 95 regardless of the OLE DB provider.*

To be comfortable with connection pooling, you need to see the connections being pooled and timed out. Unfortunately, no adequate ADO connection pool spying tools are available at the time of this writing; but you can use SQL Server's Performance Monitor, which can accurately spy on SQL Server database connections.

You can enable or disable connection pooling either in the Registry or in the connection string. The key in the Registry is `OLEDB_SERVICES`, which can be found at `HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>`. It is a bit mask that allows you to disable several OLE DB services, including connection pooling, transaction enlistment, and the cursor engine. To disable connection pooling using the connection string, include `";OLE DB Services=-2"` at the end of the connection string. To enable connection pooling for the Jet OLE DB provider, you can include `";OLE DB Services=-1"` at the end of the connection string, which enables all OLE DB services.

Persistent Recordsets

The persistent recordset is a useful feature that contributes to the briefcase model (discussed in the next section). Persistent recordsets allow you to save the contents of any recordset to a local file, which can be loaded later. In addition to aiding with the briefcase model, this feature allows developers to create true single-tier applications—you can deploy a database application without having to deploy a database. This makes for a very small footprint on your client's machine.

You can "persist" your datasets using the `SaveToFile` method:

```
ADODataSet1.SaveToFile('Local.ADTG');
```

This method saves the data and its delta in a file on your hard disk. You can reload this file using the `LoadFromFile` method, which accepts a single parameter indicating the file to load. The format of the file is Advanced Data Table Gram (ADTG), which is a proprietary Microsoft format. It does, however, have the advantage of being very efficient. If you prefer, you can save the file as XML by passing a second parameter to `SaveToFile`:

```
ADODataSet1.SaveToFile('Local.XML', pfXML);
```

However, ADO does not have a built-in XML parser (as the ClientDataSet does), so it must use the MSXML parser. Your user must either install Internet Explorer 5 or later or download the MSXML parser from the Microsoft website.

If you intend to persist your files locally in XML format, be aware of a few disadvantages:

- Saving and loading XML files is slower than saving and loading ADTG files.
- ADO's XML files (and XML files in general) are significantly larger than their ADTG counterparts (XML files are typically twice as large as their ADTG counterparts).

- ADO's XML format is specific to Microsoft, like most companies' XML implementations. This means the XML generated in ADO is not readable by the ClientDataSet and vice versa. Fortunately this problem can be overcome using Delphi's XMLTransform component, which can be used to translate between different XML structures.

If you intend to use these features solely for single-tier applications and not as part of the briefcase model, then you can use an ADODataSet component and set its `CommandType` to cmdFile and its `CommandText` to the name of the file. Doing so will save you the effort of calling `LoadFromFile` manually. However, you will still have to call `SaveToFile`. In a briefcase application this approach is too limiting, because the dataset can be used in two different modes.

The Briefcase Model

Using this knowledge of batch updates, disconnected recordsets, and persistent recordsets, you can take advantage of the *briefcase model*. The idea behind the briefcase model is that your users want to be able to use your application while they are on the road—they want to take the same application they use on their office desktops and use it on their laptops at client sites. Traditionally, the problem with this scenario is that when your users are at client sites, they are not connected to the database server, because the database server is running on the network back at their office. Consequently, there is no data on the laptop (and the data cannot be updated anyway).

This is where your newfound understanding comes in handy. Assume the application has been written; the user has requested a new briefcase enhancement, and you have to retrofit it into your existing application. You need to add a new option for your users to allow them to prepare the briefcase application by executing `SaveToFile` for every table in the database. The result is a collection of ADTG or XML files that mirror the contents of the database. These files are then copied to the laptop, where a copy of the application has previously been installed.

The application needs to be sensitive to whether it is running locally or connected to the network. You can determine this by attempting to connect to the database and seeing whether the connection fails, by detecting the presence of a local briefcase file, or by creating a flag of your own design. If the application is running in briefcase mode, then it needs to use `LoadFromFile` for each table instead of setting `Connected` to True for the ADOConnections and `Active` to True for the ADO datasets. Thereafter, the briefcase application needs to use `SaveToFile` instead of `UpdateBatch` whenever data is saved. When the user returns to the office, they need to follow an update process that loads each table from its local file, connects the dataset to the database, and applies the changes using `UpdateBatch`.

TIP *To see a complete implementation of the briefcase model, refer to the BatchUpdates example mentioned earlier.*

A Word on ADO.NET

ADO.NET is part of Microsoft's new .NET architecture—the company's redesign of application development tools to better suit the needs of web development. ADO.NET is a significant evolution of ADO. It looks at the problems of web development and addresses shortcomings of ADO's solution. The problem with ADO's solution is that it is based on COM. For one- and two-tier applications, COM imposes few problems, but in the world of web development, it is unacceptable as a transport mechanism. COM suffers from three primary problems that limit its use in web development: It (mostly) runs only on Windows, the transmission of recordsets from one process requires COM marshalling, and COM calls cannot penetrate corporate firewalls. ADO.NET's solution to all these problems is to use XML.

Some other redesign issues focus on breaking the ADO recordset into separate classes. The resulting classes are adept at solving a single problem instead of multiple problems. For example, the ADO.NET class currently called `DataSetReader` is similar to a read-only, forward-only, server-side recordset and, as such, is best suited to reading a result set very quickly. A `DataTable` is most like a disconnected, client-side recordset. A `DataRelation` shares similarities with the MSDataShape OLE DB provider. You can see that your knowledge of how ADO works is of great benefit in understanding the basic principles of ADO.NET.

What's Next?

This chapter described ActiveX Data Objects (ADO) and dbGo, the set of Delphi components for accessing the ADO interfaces. You've seen how to take advantage of Microsoft Data Access Components (MDAC) and various server engines, and I've described some of the benefits and hurdles you'll encounter in using ADO.

Chapter 16 will take you into the world of Delphi's DataSnap architecture, which is used to develop custom client and server applications in a three-tier environment. You can do this using ADO, but because this is a book about Delphi I prefer to show you the *native* solution to the problem. After we delve into DataSnap, we will continue examining Delphi's database architecture by covering the development of custom data-aware controls and dataset components.

Chapter 16

Multitier DataSnap Applications

LARGE COMPANIES OFTEN HAVE needs that are broader than applications using local database and SQL servers can meet. In the past few years, Borland Software Corporation has been addressing the needs of large corporations, and it even temporarily changed its own name to Inprise to underline this enterprise focus. The name was eventually changed back to Borland, but the focus on enterprise development remains.

Delphi targets many different technologies: three-tier architectures based on Windows NT and DCOM, TCP/IP and socket applications, and—most of all—SOAP- and XML-based web services. This chapter focuses on database-oriented multitier architectures; XML-oriented solutions will be discussed in Chapters 22 and 23, which are devoted to XML, SOAP, and web services.

Before proceeding, I should emphasize two important elements. First, the tools to support this kind of development are available only in the Enterprise version of Delphi; and second, with Delphi 7 you don't have to pay a deployment fee for DataSnap applications. You buy the development environment and then deploy your programs on as many servers as you want, without owing Borland any money. This is a very significant change (the most significant in Delphi 7) to the distribution policy of DataSnap, which used to require a per-server fee (initially very high, then significantly lowered over time). This new deployment license will certainly increase the appeal of DataSnap to developers, which is a good reason to cover it in some detail.

This chapter covers the following topics:

- Logical three-tier architecture
- The technical foundation of DataSnap
- The connection protocols and the data packets
- Delphi's support components (client-side and server-side)
- The connection broker and other extended features

One, Two, Three Levels in Delphi's History

Initially, database PC applications were client-only solutions: The program and the database files were on the same computer. From there, adventuresome programmers moved the database files onto a network file server. The client computers still hosted the application software and the entire database engine, but the database files were now accessible to several users at the same time. You can still use this type of configuration with a Delphi application and Paradox files (or, of course, Paradox itself), but the approach was much more widespread just few years ago.

The next big transition was to client/server development, embraced by Delphi since its first version. In the client/server world, the client computer requests the data from a server computer, which hosts both the database files and a database engine to access them. This architecture downplays the role of the client, but it also reduces the requirements for processing power on the client machine. Depending on how the programmers implement client/server, the server can do most (if not all) of the data processing. In this way, a powerful server can provide data services to several less powerful clients.

Naturally, there are many other reasons for using centralized database servers, such as concern for data security and integrity, simpler backup strategies, central management of data constraints, and so on. The database server is often called a SQL server, because SQL is the language most commonly used for making queries into the data; but it may also be called a RDBMS (relational database management system), reflecting the fact that the server provides tools for managing the data, such as support for backup and replication.

Of course, some applications you build may not need the benefits of a full RDBMS, so a simple client-only solution might be sufficient. On the other hand, you might need some of the robustness of a RDBMS system, but on a single, isolated computer. In this case, you can use a local version of a SQL server, such as InterBase. Traditional client/server development is done with a two-tier architecture. However, if the RDBMS is primarily performing data storage instead of data- and number-crunching, the client might contain both user interface code (formatting the output and input with customized reports, data-entry forms, query screens, and so on) and code related to managing the data (also known as *business rules*). In this case, it's generally a good idea to try to separate these two sections of the program and build a logical three-tier architecture. The term *logical* here means that there are still just two computers (that is, two physical tiers), but you've now partitioned the application into three distinct elements.

Delphi 2 introduced support for a logical three-tier architecture with data modules. As you should know by now, a *data module* is a nonvisual container for the data access components of an application (or indeed any other nonvisual components), but it often includes several handlers for database-related events. You can share a single data module among several different forms and provide different user interfaces for the same data; there might be one or more data-input forms, reports, master/detail forms, and various charting or dynamic output forms.

The logical three-tier approach solves many problems, but it also has a few drawbacks. First, you must replicate the data-management portion of the program on different client computers; doing so may hamper performance, but the bigger issue is the complexity it adds to code maintenance. Second, when multiple clients modify the same data, there's no simple way to handle the resulting update conflicts. Finally, for logical three-tier Delphi applications, you must install and configure the database engine (if any) and SQL server client library on every client computer.

The next logical step up from client/server is to move the data-module portion of the application to a separate server computer and design all the client programs to interact with it. This is exactly the purpose of remote data modules, which were introduced in Delphi 3. Remote data modules run on a server computer—generally called the *application server*. The application server in turn communicates with the DBMS (which can run on the application server or on another dedicated computer).

Therefore, the client machines don't connect to the SQL server directly, but indirectly via the application server.

At this point there is a fundamental question: Do we still need to install the database access software? The traditional Delphi client/server architecture (even with three logical tiers) requires you to install the database access on each client, which is quite troublesome when you must configure and maintain hundreds of machines. In the physical three-tier architecture, you need to install and configure the database access only on the application server, not on the client computers. Because the client programs have only user interface code and are extremely simple to install, they now fall into the category of so-called *thin clients*. To use marketing-speak, we might even call this a *zero-configuration thin-client architecture*. But let's focus on technical issues instead of marketing terminology.

The Technical Foundation of DataSnap

When Borland introduced this physical multitier architecture in Delphi, it was called MIDAS (Middle-tier Distributed Application Services). For example, Delphi 5 included the third version of this technology, MIDAS 3. Afterward Borland renamed the technology *DataSnap* and extended its capabilities.

DataSnap requires the installation of specific libraries on the server (actually the middle-tier computer), which provides your client computers with the data extracted from the SQL server database or other data sources. DataSnap does not require a SQL server for data storage. DataSnap can serve up data from a wide variety of sources, including SQL, other DataSnap servers, or data computed on the fly.

As you would expect, the client side of DataSnap is extremely thin and easy to deploy. The only file you need is `Midas.dll`, a small DLL that implements the ClientDataSet and RemoteServer components and provides the connection to the application server. As an alternative to distributing the DLL, you can embed the code of this library in your executable file by including the MidasLib unit in your `uses` statements, as discussed in Chapter 13, "Delphi's Database Architecture."

The *IAppServer* Interface

The two sides of a DataSnap application communicate using the `IAppServer` interface; this interface's definition appears in Listing 16.1. You'll seldom need to call the methods of the `IAppServer` interface directly, because Delphi includes components implementing this interface on the server side applications and components calling the interface on the client side applications. These components simplify the support of the `IAppServer` interface and at times even hide it completely. In practice, the server will make objects implementing this interface available to the client, possibly along with other custom interfaces.

LISTING 16.1: THE DEFINITION OF THE IAPPSERVER INTERFACE

```
type
  IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant;
      MaxErrors: Integer; out ErrorCount: Integer;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer;
      out RecsOut: Integer; Options: Integer; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString;
      Data: OleVariant): OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
      RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString;
      const CommandText: WideString; var Params: OleVariant;
      var OwnerData: OleVariant); safecall;
  end;
```

NOTE *A DataSnap server exposes an interface using a COM type library, a technology covered in Chapter 12, "From COM to COM+."*

The Connection Protocol

DataSnap defines only the higher-level architecture and can use different technologies for moving the data from the middle tier to the client side. DataSnap supports many different protocols, including the following:

Distributed COM (DCOM) and Stateless COM (MTS or COM+) DCOM is directly available in Windows NT/2000/XP and 98/Me, and it requires no additional run-time applications on the server. DCOM is basically an extension of COM technology that allows a client application to use server objects that exist and execute on a separate computer. The DCOM infrastructure allows you to use stateless COM objects, available in the COM+ and in the older MTS (Microsoft Transaction Server) architectures. Both COM+ and MTS provide features such as security, component management, and database transactions, and are available in Windows NT/2000/XP and in Windows 98/Me.

Due to the complexity of DCOM configuration and its problems in passing through firewalls, even Microsoft is abandoning DCOM in favor of SOAP-based solutions.

TCP/IP Sockets These are available on most systems. Using TCP/IP, you might distribute clients over the Web, where DCOM cannot be taken for granted, and you'll have far fewer configuration headaches. To use sockets, the middle-tier computer must run the `ScktSrvr.exe` application provided by Borland, a single program that can run either as an application or as a service. This program receives the client requests and forwards them to the remote data module (executing on the same server) using COM. Sockets provide no protection against failure on the client side, because the server is not informed and might not release resources when a client unexpectedly shuts down.

HTTP and SOAP The use of HTTP as a transport protocol over the Internet simplifies connections through firewalls or proxy servers (which generally don't like custom TCP/IP sockets). You need a specific web server application, `httpsrvr.dll`, which accepts client requests and creates the proper remote data modules using COM. These web connections can also use SSL security. Finally, web connections based on HTTP transport can use DataSnap object-pooling support.

NOTE *The DataSnap HTTP transport can use XML as the data packet format, enabling any platform or tool that can read XML to participate in a DataSnap architecture. This is an extension of the original DataSnap data packet format, which is also platform independent. The use of XML over HTTP is also the foundation of SOAP. There's more on SOAP in DataSnap in Chapter 23, "Web Services and SOAP."*

Until Delphi 6, you could also use CORBA (Common Object Request Broker Architecture) as a transport mechanism for DataSnap applications. Due to compatibility issues with the newer versions of Borland's VisiBroker CORBA solution, this feature has been discontinued in Delphi 7.

Finally, notice that as an extension to this architecture, you can transform the data packets into XML and deliver them to a web browser. In this case, you basically have one extra tier: the web server gets the data from the middle tier and delivers it to the client. I'll discuss this architecture, called Internet Express, in Chapter 22, "Using XML Technologies."

Providing Data Packets

The entire Delphi multitier data-access architecture centers around the idea of *data packets*. In this context, a data packet is a block of data that moves from the application server to the client or from the client back to the server. Technically, a data packet is a sort of subset of a dataset. It describes the data it contains (usually a few records of data), and it lists the names and types of the data fields. Even more important, a data packet includes the constraints—that is, the rules to be applied to the dataset. You'll typically set these constraints in the application server, and the server sends them to the client applications along with the data.

All communication between the client and the server occurs by exchanging data packets. The provider component on the server manages the transmission of several data packets within a big dataset, with the goal of responding faster to the user. As the client receives a data packet in a ClientDataSet component, the user can edit the records it contains. As mentioned earlier, during this process the client also receives and checks the constraints, which are applied during the editing operations.

When the client has updated the records and sends back a data packet, that packet is known as a *delta*. The delta packet tracks the difference between the original records and the updated ones, recording all the changes the client requested from the server. When the client asks to apply the updates to the server, it sends the delta to the server, and the server tries to apply each of the changes. I say *tries* because if a server is connected to several clients, the data may have changed already, and the update request may fail.

Because the delta packet includes the original data, the server can quickly determine if another client has already changed the data. If so, the server fires an `OnReconcileError` event, which is one of the vital elements for thin-client applications. In other words, the three-tier architecture uses an update mechanism similar to the one Delphi uses for cached updates. As you saw in Chapter 14, "Client/Server with dbExpress," the ClientDataSet manages data in a memory cache; it typically reads only a subset of the records available on the server side, loading more elements only as they're needed. When the client updates records or inserts new ones, it stores these pending changes in another local cache on the client, the *delta cache.*

The client can also save the data packets to disk and work offline, thanks to the MyBase support discussed in Chapter 13. Even error information and other data moves using the data packet protocol, so it is truly one of the foundation elements of this architecture.

NOTE *It's important to remember that data packets are protocol-independent. A data packet is merely a sequence of bytes, so anywhere you can move a series of bytes, you can move a data packet. This functionality was provided to make the architecture suitable for multiple transport protocols (including DCOM, HTTP, and TCP/IP) and for multiple platforms.*

Delphi Support Components (Client-Side)

Now that we've examined the general foundations of Delphi's three-tier architecture, let's focus on the components that support it. For developing client applications, Delphi provides the ClientDataSet component, which provides all the standard dataset capabilities and embeds the client side of the `IAppServer` interface. In this case, the data is delivered through the remote connection.

The connection to the server application is made via another component you'll also need in the client application. You should use one of the three specific connection components (available in the DataSnap page):

- The DCOMConnection component can be used on the client side to connect to a DCOM and MTS server, located either on the current computer or on another computer indicated by the `ComputerName` property. The connection is with a registered object having a given `ServerGUID` or `ServerName`.
- The SocketConnection component can be used to connect to the server via a TCP/IP socket. You should indicate the IP address or the host name, and the GUID of the server object (in the `InterceptGUID` property). This connection component has an extra property, `SupportCallbacks`, which you can disable if you are not using callbacks and want to deploy your program on Windows 95 client computers that don't have Winsock 2 installed.

NOTE *In the WebServices page, you can also find the SoapConnection component, which requires a specific type of server and will be discussed in Chapter 23.*

- The WebConnection component is used to handle an HTTP connection that can easily get through a firewall. You should indicate the URL where your copy of `httpsrvr.dll` is located and the name or GUID of the remote object on the server.

A few more client-side components were added to the DataSnap architecture in Delphi 6, mainly for managing connections:

- The ConnectionBroker component can be used as an alias of an actual connection component, which is useful when you have a single application with multiple client datasets. To change the physical connection of each dataset, you only need to change the `Connection` property of the ConnectionBroker. You can also use the events of this virtual connection component in place of those of the actual connections, so you don't have to change any code if you change the data transport technology. For the same reason, you can refer to the `AppServer` object of the ConnectionBroker instead of the corresponding property of a physical connection.

- The SharedConnection component can be used to connect to a secondary (or child) data module of a remote application, piggy-backing on an existing physical connection to the main data module. In other words, an application can connect to multiple data modules of the server with a single, shared connection.
- The LocalConnection component can be used to target a local dataset provider as the source of the data packet. The same effect can be obtained by hooking the ClientDataSet directly to the provider. However, using the LocalConnection, you can write a local application with the same code as a complete multitier application, using the `IAppServer` interface of the "fake" connection. Doing so will make the program easier to scale up, compared to a program with a direct connection.

A few other components of the DataSnap page relate to the transformation of the DataSnap data packet into custom XML formats. These components (XMLTransform, XMLTransformProvider, and XMLTransformClient) will be discussed in Chapter 22.

Delphi Support Components (Server-Side)

On the server side (actually the middle tier), you'll need to create an application or a library that embeds a remote data module, a special version of the `TDataModule` class. A second alternative is the use of a specialized remote data module for transactional COM. In the Multitier page of the New Items dialog box (obtained from the File ➢ New ➢ Other menu) there are specific wizards to create both types of remote data module.

The only specific component you need on the server side is the DataSetProvider. You need one of these components for every table or query you want to make available to the client applications, which will then use a separate ClientDataSet component for every exported dataset. The DataSetProvider was introduced in Chapter 13.

Building a Sample Application

Now you're ready to build a sample program. Doing so will let you observe some of the components I've just described in action, and will also allow you to focus on some other problems, shedding light on other pieces of the Delphi multitier puzzle. I'll build the client and application server portions of a three-tier application in two steps. The first step will simply test the technology using a minimum of elements. These programs will be very simple.

From that point, I'll add more power to the client and the application server. In each example, I'll display data from a local InterBase table using dbExpress and set up everything to allow you to test the programs on a stand-alone computer. I won't cover the steps you have to follow to install the examples on multiple computers with various technologies—that would be the subject of at least one other book.

The First Application Server

The server side of the basic example is easy to build. Simply create a new application and add a remote data module to it using the corresponding icon in the Multitier page of the Object Repository.

The Remote Data Module Wizard (see Figure 16.1) will ask you for a class name and the instancing style. When you enter a class name, such as **`AppServerOne`**, and click the OK button, Delphi will add a data module to the program. This data module will have the usual properties and events, but its class will have the following Delphi language declaration:

```
type
  TAppServerOne = class(TRemoteDataModule, IAppServerOne)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
  public
    { Public declarations }
  end;
```

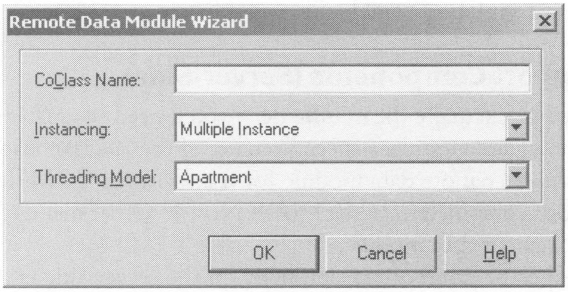


FIGURE 16.1

The Remote Data Module Wizard

CoClass Name:

Instancing: Multiple Instance

Threading Model: Apartment

In addition to inheriting from the `TRemoteDataModule` base class, this class implements the custom `IAppServerOne` interface, which derives from the standard DataSnap interface (`IAppServer`). The class also overrides the `UpdateRegistry` method to add support for enabling the socket and web transports, as you can see in the code generated by the wizard. At the end of the unit, you'll find the class factory declaration, which should be clear if you read Chapter 12:

```
initialization
  TComponentFactory.Create(ComServer, TAppServerOne,
    Class_AppServerOne, ciMultiInstance, tmApartment);
end.
```

Now you can add a dataset component to the data module (I've used the dbExpress SQLDataSet), connect it to a database and a table or query, activate it, and finally add a DataSetProvider and hook it to the dataset component. You'll obtain a DFM file like this:

```
object AppServerOne: TAppServerOne
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
  end
```

```
    object SQLDataSet1: TSQLDataSet
      SQLConnection = SQLConnection1
      CommandText = 'select * from EMPLOYEE'
    end
    object DataSetProvider1: TDataSetProvider
      DataSet = SQLDataSet1
      Constraints = True
    end
  end
```

The main form of this program is almost useless, so you can simply add a label to it indicating that it's the form of the server application. When you've built the server, you should compile it and run it once. This operation will automatically register it as an Automation server on your system, making it available to client applications. Of course, you should register the server on the computer where you want it to run, either the client or the middle tier.

The First Thin Client

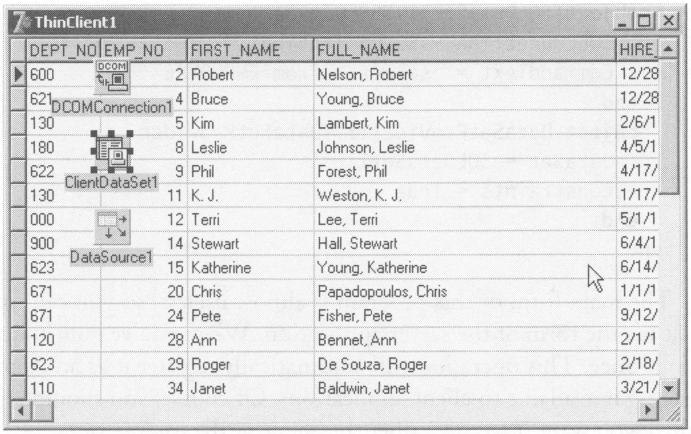
Now that you have a working server, you can build a client that will connect to it. You'll again begin with a standard Delphi application and add a DCOMConnection component to it (or the proper component for the specific type of connection you want to test). This component defines a `ComputerName` property that you'll use to specify the computer that hosts the application server. If you want to test the client and application server from the same computer, you can leave this property blank.

Once you've selected an application server computer, you can simply display the `ServerName` property's combo box list to view the available DataSnap servers. This combo box shows the servers' registered names, by default the name of the executable file of the server followed by the name of the remote data module class, as in `AppServ1.AppServerOne`. Alternatively, you can enter the GUID of the server object as the `ServerGUID` property. Delphi will automatically fill this property as you set the `ServerName` property, determining the GUID by looking it up in the Registry.

At this point, if you set the DCOMConnection component's `Connected` property to True, the server form will appear, indicating that the client has activated the server. You don't usually need to perform this operation, because the ClientDataSet component typically activates the RemoteServer component for you. I've suggested this step simply to emphasize what's happening behind the scenes.

TIP *You should generally leave the DCOMConnection component's* `Connected` *property set to False at design time, so you can open the project in Delphi even on a computer where the DataSnap server is not already registered.*

As you might expect, the next step is to add a ClientDataSet component to the form. You must connect the ClientDataSet to the `DCOMConnection1` component via the `RemoteServer` property, and thereby to one of the providers it exports. You can see the list of available providers in the `ProviderName` property, via the usual combo box. In this example, you'll be able to select only `DataSetProvider1`, because it is the only provider available in the server you've just built. This operation connects the dataset in the client's memory with the dbExpress dataset on the server. If you activate the client dataset and add a few data-aware controls (or a DBGrid), you'll immediately see the server data appear in them, as illustrated in Figure 16.2.



DEPT_NO	EMP_NO	FIRST_NAME	FULL_NAME	HIRE_
600	2	Robert	Nelson, Robert	12/28
621	4	Bruce	Young, Bruce	12/28
130	5	Kim	Lambert, Kim	2/6/1
180	8	Leslie	Johnson, Leslie	4/5/1
622	9	Phil	Forest, Phil	4/17/
130	11	K. J.	Weston, K. J.	1/17/
000	12	Terri	Lee, Terri	5/1/1
900	14	Stewart	Hall, Stewart	6/4/1
623	15	Katherine	Young, Katherine	6/14/
671	20	Chris	Papadopoulos, Chris	1/1/1
671	24	Pete	Fisher, Pete	9/12/
120	28	Ann	Bennet, Ann	2/1/1
623	29	Roger	De Souza, Roger	2/18/
110	34	Janet	Baldwin, Janet	3/21/

FIGURE 16.2

When you activate a ClientDataSet component connected to a remote data module at design time, the data from the server becomes visible as usual.

Here is the DFM file for the minimal client application, ThinCli1:

```
object Form1: TForm1
  Caption = 'ThinClient1'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object DCOMConnection1: TDCOMConnection
    ServerGUID = '{09E11D63-4A55-11D3-B9F1-00000100A27B}'
    ServerName = 'AppServ1.AppServerOne'
  end
  object ClientDataSet1: TClientDataSet
    Aggregates = <>
    Params = <>
    ProviderName = 'DataSetProvider1'
    RemoteServer = DCOMConnection1
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
end
```

Obviously, the programs for this first three-tier application are quite simple, but they demonstrate how to create a dataset viewer that splits the work between two different executable files. At this point, the client is only a viewer. If you edit the data on the client, it won't be updated on the server. To accomplish this, you'll need to add more code to the client. However, before you do that, let's add some features to the server.

Adding Constraints to the Server

When you write a traditional data module in Delphi, you can easily add some of the application logic, or business rules, by handling the dataset events and by setting field object properties and handling their events. You should avoid doing this work on the client application; instead, write your business rules on the middle tier.

In the DataSnap architecture, you can send some constraints from the server to the client and let the client program impose those constraints during the user input. You can also send field properties (such as minimum and maximum values and the display and edit masks) to the client and (using some of the data access technologies) process updates through the dataset used to access the data (or a companion UpdateSql object).

Field and Dataset Constraints

When the provider interface creates data packets to send to the client, it includes the field definitions, the table and field constraints, and one or more records (as requested by the ClientDataSet component). This implies that you can customize the middle tier and build distributed application logic by using SQL-based constraints.

The constraints you create using SQL expressions can be assigned to an entire dataset or to specific fields. The provider sends the constraints to the client along with the data, and the client applies them before sending updates back to the server. This process reduces network traffic, compared to having the client send updates back to the application server and eventually up to the SQL server, only to find that the data is invalid. Another advantage of coding the constraints on the server side is that if the business rules change, you need to update the single server application and not the many clients on multiple computers.

But how do you write constraints? You can use several properties:

- BDE datasets have a `Constraints` property, which is a collection of `TCheckConstraint` objects. Every object has a few properties, including the expression and the error message.
- Each field object defines the `CustomConstraint` and `ConstraintErrorMessage` properties. There is also an `ImportedConstraint` property for constraints imported from the SQL server.
- Each field object has a `DefaultExpression` property, which can be used locally or passed to the ClientDataSet. This is not an actual constraint; it's only a suggestion to the end user.

The next example, AppServ2, adds a few constraints to a remote data module connected to the sample EMPLOYEE InterBase database. After connecting the table to the database and creating the field objects for it, you can set the following special properties:

```
object SQLDataSet1: TSQLDataSet
  ...
  object SQLDataSet1EMP_NO: TSmallintField
    CustomConstraint = 'x > 0 and x < 10000'
    ConstraintErrorMessage =
      'Employee number must be a positive integer below 10000'
    FieldName = 'EMP_NO'
  end
  object SQLDataSet1FIRST_NAME: TStringField
    CustomConstraint = 'x <> '#39#39
    ConstraintErrorMessage = 'The first name is required'
    FieldName = 'FIRST_NAME'
    Size = 15
  end
```

```
    object SQLDataSet1LAST_NAME: TStringField
      CustomConstraint = 'not x is null'
      ConstraintErrorMessage = 'The last name is required'
      FieldName = 'LAST_NAME'
    end
  end
```

NOTE *The expression* `'x <> '#39#39` *is the DFM transposition of the string* `x <> ''`*, indicating that you don't want to have an empty string. The final constraint,* `not x is null`*, instead allows empty strings but not null values.*

Including Field Properties

You can control whether the properties of the field objects on the middle tier are sent to the ClientDataSet (and copied into the corresponding field objects of the client side) by using the poIncFieldProps value of the `Options` property of the DataSetProvider. This flag controls the download of the field properties `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, and `DisplayValues`, if they are available in the field. Here is an example of another field of the AppServ2 example with some custom properties:

```
object SQLDataSet1SALARY: TBCDField
  DefaultExpression = '10000'
  FieldName = 'SALARY'
  DisplayFormat = '#,###'
  EditFormat = '####'
  Precision = 15
  Size = 2
end
```

With this setting, you can write your middle tier the way you usually set the fields of a standard client/server application. This approach also makes it faster to move existing applications from a client/server to a multitier architecture. The main drawback of sending fields to the client is that transmitting all the extra information takes time. Turning off poIncFieldProps can dramatically improve network performance of datasets with many columns.

A server can generally filter the fields sent to the client; it does so by declaring persistent field objects with the Fields editor and omitting some of the fields. Because a field you're filtering out might be required to identify the record for future updates (if the field is part of the primary key), you can also use the field's `ProviderFlags` property on the server to send the field value to the client but make it unavailable to the ClientDataSet component (this provides some extra security, compared to sending the field to the client and hiding it there).

Field and Table Events

You can write middle-tier dataset and field event handlers as usual and let the dataset process the updates received by the client in the traditional way. This means updates are considered to be operations on the dataset, exactly as when a user is directly editing, inserting, or deleting fields locally.

This update process is requested by setting the `ResolveToDataSet` property of the `TDatasetProvider` component, again connecting either the dataset used for input or a second dataset used for the updates.

This approach is possible with datasets supporting editing operations. These include BDE, ADO, and InterBase Express datasets, but not those of the new dbExpress architecture.

With this technique, the updates are performed by the dataset, which implies a lot of control (the standard events are being triggered) but generally slower performance. Flexibility is much greater, because you can use standard coding practices. Also, porting existing local or client/server database applications, which use dataset and field events, is much more straightforward with this model. However, keep in mind that the user of the client program will receive your error messages only when the local cache (the delta) is sent back to the middle tier. Saying to the user that some data prepared half an hour ago is not valid might be a little awkward. If you follow this approach, you'll probably need to apply the updates in the cache at every `AfterPost` event on the client side.

Finally, if you decide to let the dataset and not the provider do the updates, Delphi helps you a lot in handling possible exceptions. Any exceptions raised by the middle-tier update events (for example, `OnBeforePost`) are automatically transformed by Delphi into update errors, which activate the `OnReconcileError` event on the client side (more on this event later in this chapter). No exception is shown on the middle tier, but the error travels back to the client.

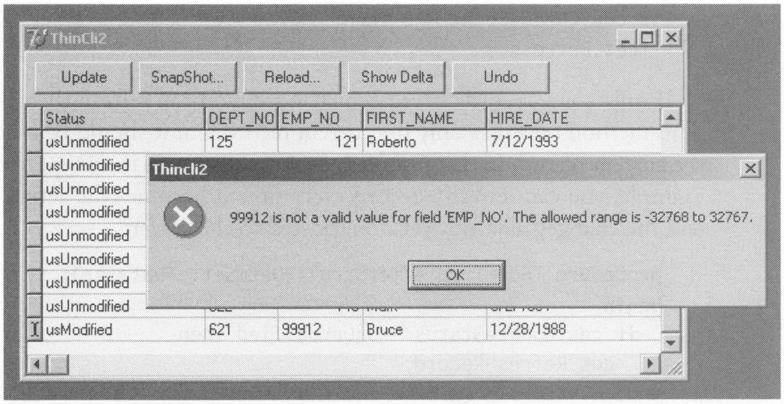
Adding Features to the Client

After adding some constraints and field properties to the server, let's return our attention to the client application. The first version was very simple, but now you can add several features to make it work well. In the ThinCli2 example, I've embedded support for checking the record status and accessing the delta information (the updates to be sent back to the server), using some of the ClientDataSet techniques already discussed in Chapter 13. The program also handles reconcile errors and supports the briefcase model.

Keep in mind that while you're using this client to edit the data locally, you'll be reminded of any failure to match the business rules of the application, which are set up on the server side using constraints. The server will also provide you with a default value for the Salary field of a new record and pass along the value of its `DisplayFormat` property. In Figure 16.3, you can see one of the error messages this client application can display, which it receives from the server. This message is displayed when you're editing the data locally, not when you send it back to the server.

FIGURE 16.3

The error message displayed by the ThinCli2 example when the employee ID is too large



Status	DEPT_NO	EMP_NO	FIRST_NAME	HIRE_DATE
usUnmodified	125	121	Roberto	7/12/1993
usUnmodified				
usUnmodified				
usUnmodified				
usUnmodified				
usUnmodified				
usUnmodified				
usUnmodified				
usModified	621	99912	Bruce	12/28/1988

99912 is not a valid value for field 'EMP_NO'. The allowed range is -32768 to 32767.

The Update Sequence

This client program includes a button to apply the updates to the server and a standard reconcile dialog. Here is a summary of the complete sequence of operations related to an update request and the possible error events:

1. The client program calls the `ApplyUpdates` method of a ClientDataSet.
2. The delta is sent to the provider on the middle tier. The provider fires the `OnUpdateData` event, where you have a chance to look at the requested changes before they reach the database server. At this point you can modify the delta, which is passed in a format compatible with the data of a ClientDataSet.
3. The provider (technically, a part of the provider called the *resolver*) applies each row of the delta to the database server. Before applying each update, the provider receives a `BeforeUpdateRecord` event. If you've set the `ResolveToDataSet` flag, this update will eventually fire local events of the dataset in the middle tier.
4. In case of a server error, the provider fires the `OnUpdateError` event (on the middle tier) and the program has a chance of fixing the error at that level.
5. If the middle-tier program doesn't fix the error, the corresponding update request remains in the delta. The error is returned to the client side at this point or after a given number of errors have been collected, depending on the value of the `MaxErrors` parameter of the `ApplyUpdates` call.
6. The delta packet with the remaining updates is sent back to the client, firing the `OnReconcileError` event of the ClientDataSet for each remaining update. In this event handler, the client program can try to fix the problem (possibly prompting the user for help), modifying the update in the delta, and later reissuing it.

Refreshing Data

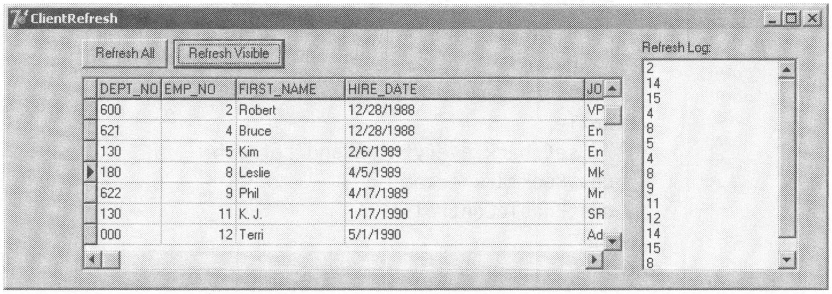
You can obtain an updated version of the data, which other users might have modified, by calling the `Refresh` method of the ClientDataSet. However, this operation can be done only if there are no pending update operations in the cache, because calling `Refresh` raises an exception when the change log is not empty:

```
if cds.ChangeCount = 0 then
  cds.Refresh;
```

If only some records have been changed, you can refresh the others by calling `RefreshRecords`. This method refreshes only the current record, but it should be used only if the user hasn't modified the current record. In this case, `RefreshRecords` leaves the unapplied changes in the change log. As an example, you can refresh a record every time it becomes the active one, unless it has been modified and the changes have not yet been posted to the server:

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);
begin
  if cds.UpdateStatus = usUnModified then
    cds.RefreshRecord;
end;
```

When the data is subject to frequent changes by many users and each user should see changes right away, you should apply any change immediately in the `AfterPost` and `AfterDelete` methods, and call `RefreshRecords` for the active record (as shown previously) or each of the records visible inside a grid. This code is part of the ClientRefresh example, connected to the AppServ2 server. For debugging purposes, the program also logs the EMP_NO field for each record it refreshes, as you can see in Figure 16.4.



DEPT_NO	EMP_NO	FIRST_NAME	HIRE_DATE	JO
600	2	Robert	12/28/1988	VP
621	4	Bruce	12/28/1988	En
130	5	Kim	2/6/1989	En
180	8	Leslie	4/5/1989	Mk
622	9	Phil	4/17/1989	Mr
130	11	K. J.	1/17/1990	SR
000	12	Terri	5/1/1990	Ad

```
2
14
15
4
8
5
4
8
9
11
12
14
15
8
```

FIGURE 16.4

The form of the ClientRefresh example, which automatically refreshes the active record and allows more extensive updates by clicking the buttons

I've done this by adding a button to the ClientRefresh example. The handler of this button moves from the current record to the first visible record of the grid and then to the last visible record. This is accomplished by noting that there are `RowCount - 1` rows visible, assuming that the first row is the fixed one hosting the field names. The program doesn't call `RefreshRecord` every time, because each movement will trigger an `AfterScroll` event with the code shown previously. The following code refreshes the visible rows, which might be triggered by a timer:

```
// protected access hack
type
  TMyGrid = class (TDBGrid)
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
  bm: TBookmarkStr;
begin
  // refresh visible rows
  cds.DisableControls;
  // start with the current row
  i := TMyGrid (DbGrid1).Row;
  bm := cds.Bookmark;
  try
    // get back to the first visible record
    while i > 1 do
    begin
      cds.Prior;
      Dec (i);
    end;
    // return to the current record
    i := TMyGrid(DbGrid1).Row;
    cds.Bookmark := bm;
    // go ahead until the grid is complete
    while i < TMyGrid(DbGrid1).RowCount do
    begin
      cds.Next;
      Inc (i);
    end;
  finally
    // set back everything and refresh
    cds.Bookmark := bm;
    cds.EnableControls;
  end;
end;
```

This approach generates a huge amount of network traffic, so you might want to trigger updates only when there are actual changes. You can implement this process by adding a callback technology to the server, so that it can inform all connected clients that a given record has changed. The client can determine whether it is interested in the change and eventually trigger the update request.

Advanced DataSnap Features

DataSnap includes many more features than I've covered up to now. Here is a quick tour of some of the more advanced features of the architecture, partially demonstrated by the AppSPlus and ThinPlus examples. Unfortunately, demonstrating every piece of functionality would turn this chapter into an entire book, so I'll limit myself to an overview.

WARNING *The ThinPlus example requires Delphi's socket server (provided in Delphi's* `bin` *folder) to run. Without this program, you'll see a socket error when the client tries to connect with the server. The plus side is that you can easily deploy the program over a network by modifying the IP address of the server in the client program.*

Besides the features discussed in the following sections, the AppSPlus and ThinPlus examples demonstrate the use of a socket connection, limited logging of events and updates on the server side, and direct fetching of a record on the client side. The last feature is accomplished with this call:

```
procedure TClientForm.ButtonFetchClick(Sender: TObject);
begin
  ButtonFetch.Caption := IntToStr (cds.GetNextPacket);
end;
```

This allows you to get more records than are required by the client user interface (the DBGrid). In other words, you can fetch records directly, without waiting for the user to scroll down in the grid. I suggest you study the details of these complex examples after reading the rest of this section.

Parametric Queries

If you want to use parameters in a query or stored procedure, then instead of building a custom solution (with a custom method call to the server), you can let Delphi help you. First define the query on the middle tier with a parameter:

```
select * from customer where Country = :Country
```

Use the `Params` property to set the type and default value of the parameter. On the client side, you can use the Fetch Params command from the ClientDataSet's shortcut menu, after connecting the component to the proper provider. At run time, you can call the equivalent `FetchParams` method of the ClientDataSet component.

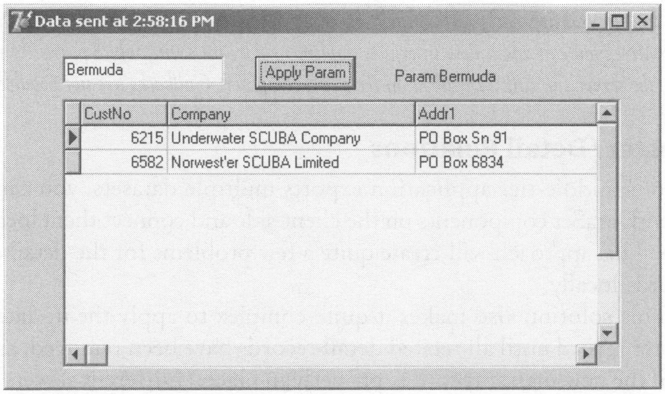
Now you can provide a local default value for the parameter by acting on the `Params` property. The value of the parameter will be sent to the middle tier when you fetch the data. The ThinPlus example refreshes the parameter with the following code:

```
procedure TFormQuery.btnParamClick(Sender: TObject);
begin
  cdsQuery.Close;
  cdsQuery.Params[0].AsString := EditParam.Text;
  cdsQuery.Open;
end;
```

You can see the secondary form of this example, which shows the result of the parametric query in a grid, in Figure 16.5. The figure also shows some custom data sent by the server, as explained in the section "Customizing the Data Packets."

FIGURE 16.5

The secondary form of the ThinPlus example, showing the data of a parametric query



Bermuda

Param Bermuda

CustNo	Company	Addr1
6215	Underwater SCUBA Company	PO Box Sn 91
6582	Norwest'er SCUBA Limited	PO Box 6834

Custom Method Calls

Because the server has a normal COM interface, you can add more methods or properties to it and call them from the client. Simply open the server's type library editor and use it as with any other COM server. In the AppSPlus example, I've added a custom `Login` method with the following implementation:

```
procedure TAppServerPlus.Login(const Name, Password: WideString);
begin
  // TODO: add actual login code...
```

```
    if Password <> Name then
      raise Exception.Create ('Wrong name/password combination received')
    else
      Query.Active := True;
    ServerForm.Add ('Login:' + Name + '/' + Password);
  end;
```

The program performs a simple test, instead of checking the name/password combination against a list of authorizations as a real application should do. Also, disabling the `Query` doesn't really work, because it can be activated by the provider; disabling the DataSetProvider is a more robust approach. The client has a simple way to access the server: the `AppServer` property of the remote connection component. Here is a sample call from the ThinPlus example, which takes place in the `AfterConnect` event of the connection component:

```
procedure TClientForm.ConnectionAfterConnect(Sender: TObject);
begin
  Connection.AppServer.Login (Edit2.Text, Edit3.Text);
end;
```

Note that you can call extra methods of the COM interface through DCOM and also using a socket-based or HTTP connection. Because the program uses the `safecall` calling convention, the exception raised on the server is automatically forwarded and displayed on the client side. This way, when a user selects the Connect check box, the event handler used to enable the client datasets is interrupted, and a user with the wrong password won't be able to see the data.

NOTE *Besides direct method calls from the client to the server, you can also implement callbacks from the server to the client. You can use this approach, for example, to notify every client of specific events. COM events are one way to do this. As an alternative, you can add a new interface, implemented by the client, which passes the implementation object to the server. This way, the server can call the method on the client computer. Callbacks are not possible with HTTP connections, though.*

Master/Detail Relations

If your middle-tier application exports multiple datasets, you can retrieve them using multiple ClientDataSet components on the client side and connect them locally to form a master/detail structure. This approach will create quite a few problems for the detail dataset unless you retrieve all the records locally.

This solution also makes it quite complex to apply the updates; you cannot usually cancel a master record until all related detail records have been removed, and you cannot add detail records until the new master record is properly in place. (Different servers handle this situation differently, but in most cases where a foreign key is used, this is the standard behavior.) To solve this problem, you can write complex code on the client side to update the records of the two tables according to the specific rules.

A completely different approach is to retrieve a single dataset that already includes the detail as a dataset field, a field of type `TDataSetField`. To accomplish this, you need to set up the master/detail relation on the server application:

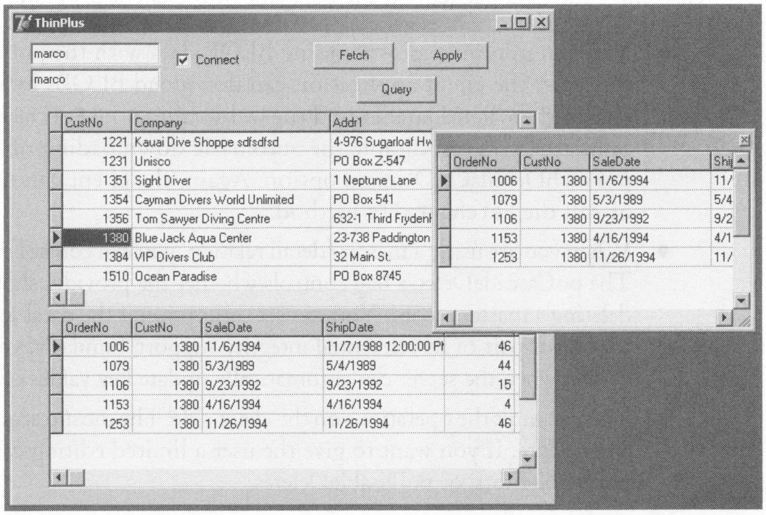
```
object TableCustomer: TTable
  DatabaseName = 'DBDEMOS'
```

```
    TableName = 'customer.db'
  end
  object TableOrders: TTable
    DatabaseName = 'DBDEMOS'
    MasterFields = 'CustNo'
    MasterSource = DataSourceCust
    TableName = 'ORDERS.DB'
  end
  object DataSourceCust: TDataSource
    DataSet = TableCustomer
  end
  object ProviderCustomer: TDataSetProvider
    DataSet = TableCustomer
  end
```

On the client side, the detail table will show up as an extra field of the ClientDataSet, and the DBGrid control will display it as an extra column with an ellipsis button. Clicking the button will display a secondary form with a grid presenting the detail table (see Figure 16.6). If you need to build a flexible user interface on the client, you can then add a secondary ClientDataSet connected to the dataset field of the master dataset, using the `DataSetField` property. Simply create persistent fields for the main ClientDataSet and then hook up the property:

```
  object cdsDet: TClientDataSet
    DataSetField = cdsTableOrders
  end
```

With this setting, you can show the detail dataset in a separate DBGrid placed as usual in the form (the bottom grid of Figure 16.6) or any other way you like. Note that with this structure, the updates relate only to the master table, and the server should handle the proper update sequence even in complex situations.



marco

marco

☑ Connect

CustNo	Company	Addr1
1221	Kauai Dive Shoppe sdfsdfsd	4-976 Sugarloaf Hw
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenh
1380	Blue Jack Aqua Center	23-738 Paddington
1384	VIP Divers Club	32 Main St.
1510	Ocean Paradise	PO Box 8745

OrderNo	CustNo	SaleDate	Shi
1006	1380	11/6/1994	11/
1079	1380	5/3/1989	5/4
1106	1380	9/23/1992	9/2
1153	1380	4/16/1994	4/1
1253	1380	11/26/1994	11/

OrderNo	CustNo	SaleDate	ShipDate	
1006	1380	11/6/1994	11/7/1988 12:00:00 PI	46
1079	1380	5/3/1989	5/4/1989	44
1106	1380	9/23/1992	9/23/1992	15
1153	1380	4/16/1994	4/16/1994	4
1253	1380	11/26/1994	11/26/1994	46

FIGURE 16.6

The ThinPlus example shows how a dataset field can either be displayed in a grid in a floating window or extracted by a ClientDataSet and displayed in a second form. You'll generally do one of the two things, not both!

Using the Connection Broker

I've already mentioned that the ConnectionBroker component can be helpful in case you might want to change the physical connection used by many ClientDataSet components of a single program. By hooking each ClientDataSet to the ConnectionBroker, you can change the physical connection of all the ClientDataSets by updating the physical connection of the broker.

The ThinPlus example uses these settings:

```
object Connection: TSocketConnection
  ServerName = 'AppSPlus.AppServerPlus'
  AfterConnect = ConnectionAfterConnect
  Address = '127.0.0.1'
end
object ConnectionBroker1: TConnectionBroker
  Connection = Connection
end
object cds: TClientDataSet
  ConnectionBroker = ConnectionBroker1
end
// in the secondary form
object cdsQuery: TClientDataSet
  ConnectionBroker = ClientForm.ConnectionBroker1
end
```

That's basically all you have to do. To change the physical connection, drop a new DataSnap connection component to the main form and set the `Connection` property of the broker to it.

More Provider Options

I've already mentioned the `Options` property of the DataSetProvider component, noting that you can use it to add the field properties to the data packet. There are several other options you can use to customize the data packet and the behavior of the client program. Here is a short list:

- You can minimize downloading BLOB data with the poFetchBlobsOnDemand option. In this case, the client application can download BLOBs by setting the `FetchOnDemand` property of the ClientDataSet to True or by calling the `FetchBlobs` method for specific records. Similarly, you can disable the automatic downloading of detail records by setting the poFetchDetailsOnDemand option. Again, the client can use the `FetchOnDemand` property or call the `FetchDetails` method.
- When you are using a master/detail relation, you can control cascades with either of two options. The poCascadeDeletes flag controls whether the provider should delete detail records before deleting a master record. You can set this option if the database server performs cascaded deletes for you as part of its referential integrity support. Similarly, you can set the poCascadeUpdates option when the server can automatically update key values of a master/detail relation.
- You can limit the operations on the client side. The most restrictive option, poReadOnly, disables any update. If you want to give the user a limited editing capability, use poDisableInserts, poDisableEdits, or poDisableDeletes.

- You can use poAutoRefresh to resend to the client a copy of the records the client has modified; doing so is useful in case other users have simultaneously made other, nonconflicting changes. You can also specify the poPropogateChanges option to send back to the client changes done in the `BeforeUpdateRecord` or `AfterUpdateRecord` event handler. This option is also handy when you are using autoincrement fields, triggers, and other techniques that modify data on the server or middle tier beyond the changes requested from the client tier.
- Finally, if you want the client to drive the operations, you can enable the poAllowCommandText option. It lets you set the SQL query or table name of the middle tier from the client, using the `GetRecords` or `Execute` method.

The Simple Object Broker

The SimpleObjectBroker component provides an easy way to locate a server application among several server computers. You provide a list of available computers, and the client will try each of them in order until it finds one that is available.

Moreover, if you enable the `LoadBalanced` property, the component will randomly choose one of the servers; when many clients use the same configuration, the connections will be automatically distributed among the multiple servers. If this seems like a "poor man's" object broker, consider that some highly expensive load-balancing systems don't offer much more than this.

Object Pooling

When multiple clients connect to your server at the same time, you have two options. The first is to create a remote data module object for each of them and let each request be processed in sequence (the default behavior for a COM server with the ciMultiInstance style). Alternatively, you can let the system create a different instance of the server application for every client (ciSingleInstance). This approach requires more resources and more SQL server connections (and possibly licenses).

An alternate approach is offered by DataSnap's support for object pooling. All you need to do to request this feature is add a call to `RegisterPooled` in the overridden `UpdateRegistry` method. Combined with the stateless support built in to this architecture, the pooling capability allows you to share some middle-tier objects among a much larger number of clients. A pooling mechanism is built in to COM+, but DataSnap makes it available for HTTP and socket-based connections as well.

The users on the client computers will spend most of their time reading data and typing in updates, and they generally don't continue asking for data and sending updates. When the client is not calling a method of the middle-tier object, this same remote data module can be used for another client. Being stateless, every request reaches the middle tier as a brand-new operation, even when a server is dedicated to a specific client.

Customizing the Data Packets

There are many ways to include custom information within the data packet handled by the `IAppServer` interface. The simplest is to handle the `OnGetDataSetProperties` event of the provider. This event has a `Sender` parameter, a dataset parameter indicating where the data is coming from, and an `OleVariant` array `Properties` parameter, in which you can place the extra information. You need to define one variant array for each extra property and include the name of the extra property, its value, and whether you want the data to return to the server along with the update delta (the `IncludeInDelta` parameter).

Of course, you can pass properties of the related dataset component, but you can also pass any other value (extra fake properties). In the AppSPlus example, I pass to the client the time the query was executed and its parameters:

```
procedure TAppServerPlus.ProviderQueryGetDataSetProperties(
  Sender: TObject; DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['Time', Now, True]);
  Properties[1] := VarArrayOf(['Param', Query.Params[0].AsString, False]);
end;
```

On the client side, the ClientDataSet component has a `GetOptionalParameter` method to retrieve the value of the extra property with the given name. The ClientDataSet also has the `SetOptionalParameter` method to add more properties to the dataset. These values will be saved to disk (in the briefcase model) and eventually sent back to the middle tier (by setting the `IncludeInDelta` member of the variant array to True). Here is a simple example of the retrieval of the dataset in the previous code:

```
Caption := 'Data sent at ' + TimeToStr (TDateTime (
  cdsQuery.GetOptionalParam('Time')));
Label1.Caption := 'Param ' + cdsQuery.GetOptionalParam('Param');
```

The effect of this code was visible in Figure 16.5. An alternative and more powerful approach for customizing the data packet sent to the client is to handle the `OnGetData` event of the provider, which receives the outgoing data packet in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client. For example, you might encode some of the data or filter out sensitive records.

What's Next?

Borland originally introduced its multitier technology in Delphi 3 and has kept extending it from version to version. In addition to further updates and the change of the MIDAS name to DataSnap, Delphi 6 saw the introduction of SOAP support, introducing an alternate and extended architecture for multitier applications. We'll fully explore this topic in Chapter 23. Delphi 7, on the other hand, cut back on CORBA support.

However, with the introduction of a new licensing scheme (basically, *free deployment*), Delphi 7 has paved the way for increased adoption of this technology. This is particularly true of DataSnap's SOAP variant, but socket connections also provide a very good balance of data transfer efficiency and ease of deployment.

For the moment, we'll continue with database programming, discussing data-aware controls and custom datasets. In the final part of the book we'll explore sockets, Internet programming, and SOAP, so you'll have a complete picture of possible multitier architectures based on Delphi—not to mention the availability of third-party tools providing features similar to DataSnap.

Chapter 17

Writing Database Components

IN CHAPTER 9, "WRITING Delphi Components," we explored the development of Delphi components in depth. Now that I've discussed database programming, we can get back to the earlier topic and focus on the development of database-related components.

There are basically two families of such components: data-aware controls you can use to present the data of a field or an entire record to the users of a program; and dataset components you can define to provide data to existing data-aware controls, reading the data from a database or any other data source. In this chapter, I'll cover both topics.

This chapter covers the following:

- Data-aware components: the data link
- Field-oriented data-aware controls
- Data-aware TrackBar and ProgressBar
- Record-oriented data-aware controls
- A record viewer
- Building custom datasets
- Saving a dataset to a local stream

The Data Link

When you write a Delphi database program, you generally connect some data-aware controls to a DataSource component, and then connect the DataSource component to a dataset. The connection between the data-aware control and the DataSource is called a *data link* and is represented by an object of class `TDataLink` or descendant. The data-aware control creates and manages this object and represents its only connection to the data. From a more practical perspective, to make a component data-aware, you need to add a data link to it and surface some of the properties of this internal object, such as the `DataSource` and `DataField` properties.

Delphi uses the DataSource and DataLink objects for bidirectional communication. The dataset uses the connection to notify the data-aware controls that new data is available (because the dataset has been activated, or the current record has changed, and so on). Data-aware controls use the connection to ask for the current value of a field or to update it, notifying the dataset of this event.

The relations among all these components are complicated by the fact that some of the connections can be one-to-many. For example, you can connect multiple data sources to the same dataset, you generally have multiple data links to the same data source (simply because you need one link for every data-aware component), and in most cases you connect multiple data-aware controls to each data source.

The *TDataLink* Class

We'll work for much of this chapter with `TDataLink` and its derived classes, which are defined in the DB unit. This class has a set of protected virtual methods, which have a role similar to events. They are "almost-do-nothing" methods you can override in a specific subclass to intercept user operations and other data-source events. Here is a list, extracted from the class's source code:

```
type
  TDataLink = class(TPersistent)
  protected
    procedure ActiveChanged; virtual;
    procedure CheckBrowseMode; virtual;
    procedure DataSetChanged; virtual;
    procedure DataSetScrolled(Distance: Integer); virtual;
    procedure FocusControl(Field: TFieldRef); virtual;
    procedure EditingChanged; virtual;
    procedure LayoutChanged; virtual;
    procedure RecordChanged(Field: TField); virtual;
    procedure UpdateData; virtual;
```

All these virtual methods are called by the `DataEvent` private method, which is a sort of window procedure for a data source, triggered by several data events (see the `TDataEvent` enumeration). These events originate in the dataset, fields, or data source, and are generally applied to a dataset. The `DataEvent` method of the dataset component dispatches the events to the connected data sources. Each data source calls the `NotifyDataLinks` method to forward the event to each connected data link, and then the data source triggers its own `OnDataChange` or `OnUpdateData` event.

Derived Data Link Classes

The `TDataLink` class is not technically an abstract class, but you'll seldom use it directly. When you need to create data-aware controls, you'll need to use one of its derived classes or derive a new one yourself. The most important class derived from `TDataLink` is the `TFieldDataLink` class, which is used by data-aware controls that relate to a single field of the dataset. Most data-aware controls fall into this category, and the `TFieldDataLink` class solves the most common problems of this type of component.

All the table- or record-oriented data-aware controls define specific subclasses of `TDataLink`, as we'll do later. The `TFieldDataLink` class has a list of events corresponding to the virtual methods of the base class it overrides. This makes the class simpler to customize, because you can use event handlers instead of having to inherit a new class from it. Here's an example of an overridden method, which fires the corresponding event, if available:

```
procedure TFieldDataLink.ActiveChanged;
begin
  UpdateField;
  if Assigned(FOnActiveChange) then FOnActiveChange(Self);
end;
```

The `TFieldDataLink` class also contains the `Field` and `FieldName` properties that let you connect the data-aware control to a specific field of the dataset. The link keeps a reference to the current visual component, using the `Control` property.

Writing Field-Oriented Data-Aware Controls

Now that you understand the theory of how the data link classes work, let's begin building some data-aware controls. The first two examples are data-aware versions of the ProgressBar and TrackBar common controls. You can use the first to display a numeric value, such as a percentage, in a visual way. You can use the second to allow a user to change the numeric value.

NOTE *The code for all of the components built in this chapter is in the MdDataPack folder, which also includes a similarly named package for installing them all. Other folders include sample programs that use these components.*

A Read-Only ProgressBar

A data-aware version of the ProgressBar control is a relatively simple case of a data-aware control, because it is read-only. This component is derived from the version that's not data-aware and adds a few properties of the data link object it encapsulates:

```
type
  TMdDbProgress = class(TProgressBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
  protected
    // data link event handler
    procedure DataChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

As with every data-aware component that connects to a single field, this control makes available the `DataSource` and `DataField` properties. There is little code to write; simply export the properties from the internal data link object, as follows:

```
function TMdDbProgress.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TMdDbProgress.SetDataField (Value: string);
begin
  FDataLink.FieldName := Value;
end;

function TMdDbProgress.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TMdDbProgress.SetDataSource (Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;

function TMdDbProgress.GetField: TField;
begin
  Result := FDataLink.Field;
end;
```

Of course, to make this component work, you must create and destroy the data link when the component itself is created or destroyed:

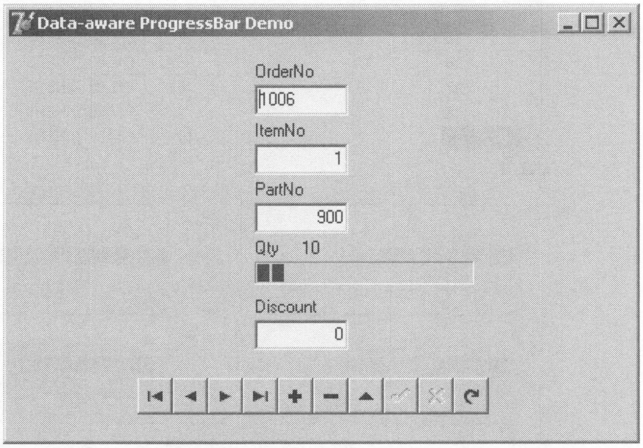
```
constructor TMdDbProgress.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := Self;
  FDataLink.OnDataChange := DataChange;
end;

destructor TMdDbProgress.Destroy;
begin
  FDataLink.Free;
  FDataLink := nil;
  inherited Destroy;
end;
```

In the preceding constructor, notice that the component installs one of its own methods as an event handler for the data link. This is where the component's most important code resides. Every time the data changes, you modify the output of the progress bar to reflect the value of the current field:

```
procedure TMdDbProgress.DataChange (Sender: TObject);
begin
  if FDataLink.Field is TNumericField then
    Position := FDataLink.Field.AsInteger
  else
    Position := Min;
end;
```

Following the convention of the VCL data-aware controls, if the field type is invalid, the component doesn't display an error message—it disables the output. Alternatively, you might want to check the field type when the `SetDataField` method assigns it to the control.

In Figure 17.1 you can see an example of the DbProgr application's output, which uses both a label and a progress bar to display an order's quantity information. Thanks to this visual clue, you can step through the records and easily spot orders for many items. One obvious benefit to this component is that the application contains almost no code, because all the important code is in the MdProgr unit that defines the component.



1006

1

900

10

0

FIGURE 17.1
The data-aware ProgressBar in action in the DbProgr example

As you've seen, a read-only data-aware component is not difficult to write. However, it becomes extremely complex to use such a component inside a DBCtrlGrid container.

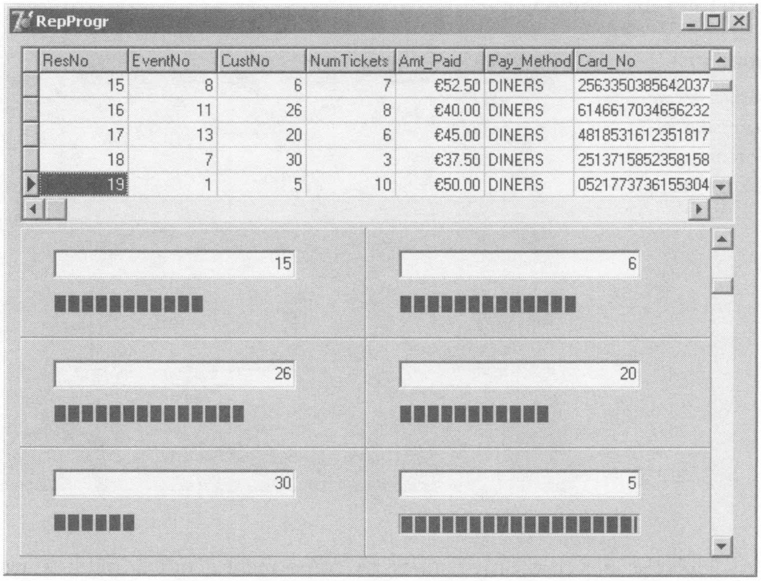
NOTE *If you remember the discussion of the* `Notification` *method in Chapter 9, you might wonder what happens if the data source referenced by the data-aware control is destroyed. The good news is that the data source has a destructor that removes itself from its own data links. So, there is no need for a* `Notification` *method for data-aware controls, although you'll see books and articles suggesting it, and VCL includes plenty of this useless code.*

REPLICABLE DATA-AWARE CONTROLS

Extending a data-aware control to support its use inside a DBCtrlGrid component is complex and not well documented. You can find a complete replicable version of the progress bar in the MdRepPr unit of the MdDataPack package and an example of its use in the `RepProgr` folder, along with an HTML file describing its development. The DBCtrlGrid component has a peculiar behavior: It displays on screen multiple versions of the same physical control, using smoke and mirrors. The grid can attach the control to a data buffer other than the current record and redirects the control paint operations to another portion of the monitor.

In short, to appear in the DBCtrlGrid, a component's csReplicatable control style must be set; this flag indicates that your component supports being hosted by a control grid. First, the component must respond to the `cm_GetDataLink` Delphi message and return a pointer to the data link, so that the control grid can use and change it. Second, it needs a custom `Paint` method to draw the output in the appropriate canvas object, which is provided in a parameter of the `wm_Paint` message if the `ControlState` property's csPaintCopy flag is set.

The example code is involved, and the DBCtrlGrid component is not heavily used, so I decided not to give you full details here; you can find the full code and more information in the source code. Here's the output of a test program that uses this component:



ResNo	EventNo	CustNo	NumTickets	Amt_Paid	Pay_Method	Card_No
15	8	6	7	€52.50	DINERS	2563350385642037
16	11	26	8	€40.00	DINERS	6146617034656232
17	13	20	6	€45.00	DINERS	4818531612351817
18	7	30	3	€37.50	DINERS	2513715852358158
19	1	5	10	€50.00	DINERS	0521773736155304

15

6

26

20

30

5

A Read-Write TrackBar

The next step is to write a component that allows a user to modify the data in a database, not just browse it. The overall structure of this type of component isn't very different from the previous version, but there are a few extra elements. In particular, when the user begins interacting with the component, the code should put the dataset into edit mode and then notify the dataset that the data has changed. The dataset will then use a `FieldDataLink` event handler to ask for the updated value.

To demonstrate how you can create a data-aware component that modifies the data, I extended the TrackBar control. This isn't the simplest example, but it demonstrates several important techniques.

Here's the definition of the component's class (from the MdTrack unit of the MdDataPack package):

```
type
  TMdDbTrack = class(TTrackBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
    procedure CMExit(var Message: TCMExit); message CM_EXIT;
  protected
    // data link event handlers
    procedure DataChange (Sender: TObject);
    procedure UpdateData (Sender: TObject);
    procedure ActiveChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

Compared to the read-only data-aware control you built earlier, this class is more complex, because it has three message handlers, including component notification handlers, and two new event handlers for the data link. The component installs these event handlers in the constructor, which also disables the component:

```
constructor TMdDbTrack.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := Self;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData:= UpdateData;
  FDataLink.OnActiveChange := ActiveChange;
  Enabled := False;
end;
```

The get and set methods and the `DataChange` event handler are similar to those in the `TMdDbProgress` component. The only difference is that whenever the data source or data field changes, the component checks the current status to see whether it should enable itself:

```
procedure TMdDbTrack.SetDataSource (Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;
```

This code tests three conditions: the data link should be active, the link should refer to an actual field, and the field shouldn't be read-only.

When the user changes the field, the component should consider that the field name might be invalid; to test for this condition, the component uses a `try/finally` block:

```
procedure TMdDbTrack.SetDataField (Value: string);
begin
  try
    FDataLink.FieldName := Value;
  finally
    Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
      not FDataLink.Field.ReadOnly;
  end;
end;
```

The control executes the same test when the dataset is enabled or disabled:

```
procedure TMdDbTrack.ActiveChange (Sender: TObject);
begin
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;
```

The most interesting portion of this component's code is related to its user interface. When a user begins moving the scroll thumb, the component puts the dataset into edit mode, lets the base class update the thumb position, and alerts the data link (and therefore the data source) that the data has changed. Here's the code:

```
procedure TMdDbTrack.CNHScroll(var Message: TWMHScroll);
begin
  // enter edit mode
  FDataLink.Edit;
  // update data
  inherited;
  // let the system know
  FDataLink.Modified;
end;

procedure TMdDbTrack.CNVScroll(var Message: TWMVScroll);
begin
  // enter edit mode
  FDataLink.Edit;
```

```
    // update data
    inherited;
    // let the system know
    FDataLink.Modified;
  end;
```

When the dataset needs new data—for example, to perform a `Post` operation—it requests it from the component via the `TFieldDataLink` class's `OnUpdateData` event:

```
  procedure TMdDbTrack.UpdateData (Sender: TObject);
  begin
    if FDataLink.Field is TNumericField then
      FDataLink.Field.AsInteger := Position;
  end;
```

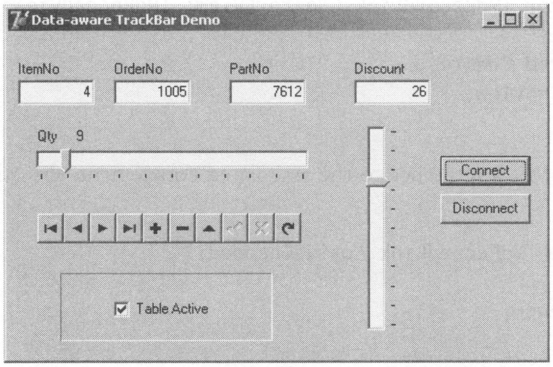
If the proper conditions are met, the component updates the data in the proper table field. Finally, if the component loses the input focus, it should force a data update (if the data has changed) so that any other data-aware components showing the value of that field will display the correct value as soon as the user moves to a different field. If the data hasn't changed, the component doesn't bother updating the data in the table. This is the standard `CMExit` code for components used by VCL and borrowed for this component:

```
  procedure TMdDbTrack.CMExit(var Message: TCMExit);
  begin
    try
      FDataLink.UpdateRecord;
    except
      SetFocus;
      raise;
    end;
    inherited;
  end;
```

A demo program is available for testing this component; you can see its output in Figure 17.2. The DbTrack program contains a check box to enable and disable the table, the visual components, and a couple of buttons you can use to detach the vertical TrackBar component from the field it relates to. I placed these on the form to test enabling and disabling the track bar.

FIGURE 17.2

The DbTrack example's track bars let you enter data in a database table. The check box and buttons test the enabled status of the components.



Data-aware TrackBar Demo

ItemNo: 4

OrderNo: 1005

PartNo: 7612

Discount: 26

Qty 9

☑ Table Active

Creating Custom Data Links

The data-aware controls I've built up to this point all refer to specific fields of the dataset, so I used a `TFieldDataLink` object to establish the connection with a data source. Now let's build a data-aware component that works with a dataset as a whole: a record viewer.

Delphi's database grid shows the value of several fields and several records simultaneously. My record viewer component lists all the fields of the current record, using a customized grid. This example will show you how to build a customized grid control and a custom data link to go with it.

A Record Viewer Component

In Delphi there are no data-aware components that manipulate multiple fields of a single record without displaying other records. The only two components that display multiple fields from the same table are the DBGrid and the DbCtrlGrid, which generally display multiple fields and multiple records.

The record viewer component I'll describe in this section is based on a two-column grid; the first column displays the table's field names, and the second column displays the corresponding field values. The number of rows in the grid corresponds to the number of fields, with a vertical scroll bar in case they can't fit in the visible area.

The data link you need in order to build this component is a class connected only to the record viewer component and declared directly in the implementation portion of its unit. This is the same approach used by VCL for some specific data links. Here's the definition of the new class:

```
type
  TMdRecordLink = class (TDataLink)
  private
    RView: TMdRecordView;
  public
    constructor Create (View: TMdRecordView);
    procedure ActiveChanged; override;
    procedure RecordChanged (Field: TField); override;
  end;
```

As you can see, the class overrides the methods related to the principal event—in this case, the activation and data (or record) change. Alternatively, you could export events and then let the component handle them, as the `TFieldDataLink` does.

The constructor requires the associated component as its only parameter:

```
constructor TMdRecordLink.Create (View: TMdRecordView);
begin
  inherited Create;
  RView := View;
end;
```

After you store a reference to the associated component, the other methods can operate on it directly:

```
procedure TMdRecordLink.ActiveChanged;
var
  I: Integer;
```

```
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // repaint all...
  RView.Invalidate;
end;

procedure TMdRecordLink.RecordChanged;
begin
  inherited;
  // repaint all...
  RView.Invalidate;
end;
```

The record link code is simple. Most of the difficulties in building this example result from the use of a grid. To avoid dealing with useless properties, I derived the record viewer grid from the `TCustomGrid` class. This class incorporates much of the code for grids, but most of its properties, events, and methods are protected. For this reason, the class declaration is quite long, because it needs to publish many existing properties. Here is an excerpt (excluding the base class properties):

```
type
  TMdRecordView = class(TCustomGrid)
  private
    // data-aware support
    FDataLink: TDataLink;
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
  protected
    // redefined TCustomGrid methods
    procedure DrawCell (ACol, ARow: Longint; ARect: TRect;
      AState: TGridDrawState); override;
    procedure ColWidthsChanged; override;
    procedure RowHeightsChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure SetBounds (ALeft, ATop, AWidth, AHeight: Integer); override;
    // public parent properties (omitted...)
  published
    // data-aware properties
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    // published parent properties (omitted...)
  end;
```

In addition to redeclaring the properties to publish them, the component defines a data link object and the `DataSource` property. There's no `DataField` property for this component, because it refers to an entire record. The component's constructor is very important. It sets the values of many unpublished properties, including the grid options:

```
constructor TMdRecordView.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TMdRecordLink.Create (self);
  // set numbers of cells and fixed cells
  RowCount := 2; // default
  ColCount := 2;
  FixedCols := 1;
  FixedRows := 0;
  Options:= [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goRowSizing];
  DefaultDrawing := False;
  ScrollBars := ssVertical;
  FSaveCellExtents := False;
end;
```

The grid has two columns (one of them fixed) and no fixed rows. The fixed column is used for resizing each row of the grid. Unfortunately, a user cannot drag the fixed row to resize the columns, because you can't resize fixed elements, and the grid already has a fixed column.

NOTE *An alternative approach could be to have an extra empty column, like the DBGrid control. You could resize the two other columns after adding a fixed row. Overall, though, I prefer my implementation.*

I used an alternative approach to resize the columns. The first column (holding the field names) can be resized either using programming code or visually at design time, and the second column (holding the values of the fields) is resized to use the remaining area of the component:

```
procedure TMdRecordView.SetBounds (ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited;
  ColWidths [1] := ClientWidth - ColWidths[0];
end;
```

This resizing takes place when the component size changes and when either of the columns change. With this code, the `DefaultColWidth` property of the component becomes, in practice, the fixed width of the first column.

After everything has been set up, the key method of the component is the overridden `DrawCell` method, detailed in Listing 17.1. In this method, the control displays the information about the fields and their values. It needs to draw three things. If the data link is not connected to a data source, the grid displays an *empty element* sign (*[]*). When drawing the first column, the record viewer shows the `DisplayName` of the field, which is the same value used by the DBGrid for the heading. When drawing the second column, the component accesses the textual representation of the field value, extracted with the `DisplayText` property (or with the `AsString` property for memo fields).

LISTING 17.1: THE *DRAWCELL* METHOD OF THE CUSTOM RECORDVIEW COMPONENT

```
procedure TMdRecordView.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
var
  Text: string;
  CurrField: TField;
  Bmp: TBitmap;
begin
  CurrField := nil;
  Text := '[]'; // default
  // paint background
  if (ACol = 0) then
    Canvas.Brush.Color := FixedColor
  else
    Canvas.Brush.Color := Color;
  Canvas.FillRect (ARect);
  // leave small border
  InflateRect (ARect, -2, -2);
  if (FDataLink.DataSource <> nil) and FDataLink.Active then
  begin
    CurrField := FDataLink.DataSet.Fields[ARow];
    if ACol = 0 then
      Text := CurrField.DisplayName
    else if CurrField is TMemoField then
      Text := TMemoField (CurrField).AsString
    else
      Text := CurrField.DisplayText;
  end;
  if (ACol = 1) and (CurrField is TGraphicField) then
  begin
    Bmp := TBitmap.Create;
    try
      Bmp.Assign (CurrField);
      Canvas.StretchDraw (ARect, Bmp);
    finally
      Bmp.Free;
    end;
  end
  else if (ACol = 1) and (CurrField is TMemoField) then
  begin
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
      dt_WordBreak or dt_NoPrefix)
  end
  else // draw single line vertically centered
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
      dt_vcenter or dt_SingleLine or dt_NoPrefix);
  if gdFocused in AState then
    Canvas.DrawFocusRect (ARect);
end;
```

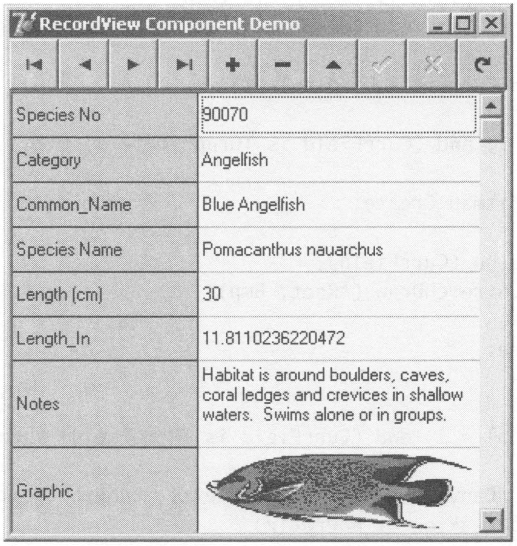
In the final portion of the method, the component considers memo and graphic fields. If the field is a `TMemoField`, the `DrawText` function call doesn't specify the dt_SingleLine flag, but uses dt_WordBreak flag to wrap the words when there's no more room. For a graphic field, the component uses a completely different approach, assigning the field image to a temporary bitmap and then stretching it to fill the surface of the cell.

Notice that the component sets the `DefaultDrawing` property to False, so it's also responsible for drawing the background and the focus rectangle, as it does in the `DrawCell` method. The component also calls the `InflateRect` API function to leave a small area between the cell border and the output text. The output is produced by calling another Windows API function, `DrawText`, which centers the text vertically in its cell.

This drawing code works both at run time, as you can see in Figure 17.3, and at design time. The output may not be perfect, but this component can be useful in many cases. To display the data for a single record, instead of building a custom form with labels and data-aware controls, you can easily use this record viewer grid. It's important to remember that the record viewer is a read-only component. It's possible to extend it to add editing capabilities (they're already part of the `TCustomGrid` class); however, instead of adding this support, I decided to make the component more complete by adding support for displaying BLOB fields.

FIGURE 17.3

The ViewGrid example demonstrates the output of the RecordView component, using Borland's sample BioLife database table.



Species No	90070
Category	Angelfish
Common_Name	Blue Angelfish
Species Name	Pomacanthus nauarchus
Length (cm)	30
Length_In	11.8110236220472
Notes	Habitat is around boulders, caves, coral ledges and crevices in shallow waters. Swims alone or in groups.
Graphic	

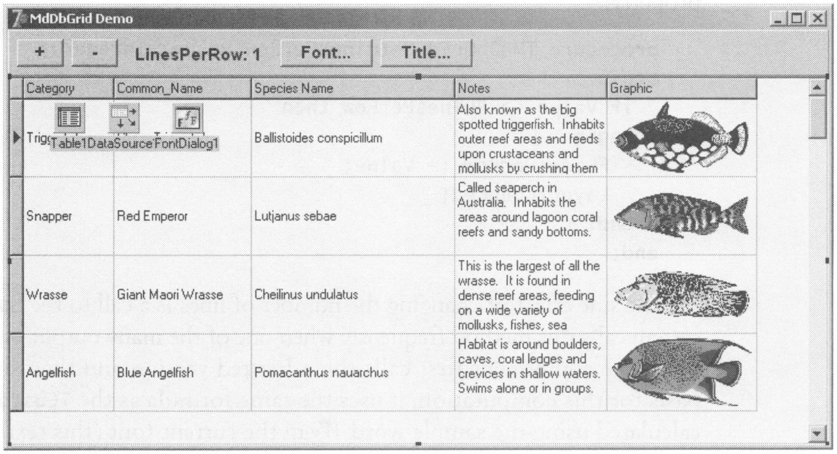
To improve the graphical output, the control makes the lines for BLOB fields twice as high as those for plain text fields. This operation is accomplished when the dataset connected to the data-aware control is activated. The data link's `ActiveChanged` method is also triggered by the `RowHeightsChanged` methods connected to the `DefaultRowHeight` property of the base class:

```
procedure TMdRecordLink.ActiveChanged;
var
  I: Integer;
begin
  // set number of rows
  RView.RowCount := DataSet.FieldCount;
  // double the height of memo and graphics
  for I := 0 to DataSet.FieldCount - 1 do
    if DataSet.Fields [I] is TBlobField then
      RView.RowHeights [I] := RView.DefaultRowHeight * 2;
  // repaint all...
  RView.Invalidate;
end;
```

At this point, you stumble into a minor problem. In the `DefineProperties` method, the `TCustomGrid` class saves the values of the `RowHeights` and `ColHeights` properties. You could disable this streaming by overriding the method and not calling `inherited` (which is generally a bad technique), but you can also toggle the `FSaveCellExtents` protected field to disable this feature (as I've done in the component's code).

Customizing the DBGrid Component

In addition to writing new, custom, data-aware components, Delphi programmers commonly customize the DBGrid control. The goal for the next component is to enhance the DBGrid with the same kind of custom output I used for the RecordView component, directly displaying graphic and memo fields. To do this, the grid needs to make the row height resizable, to allow space for graphics and a reasonable amount of text. You can see an example of this grid at design time in Figure 17.4.



Category	Common_Name	Species Name	Notes	Graphic
Trig		Ballistoides conspicillum	Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them	
Snapper	Red Emperor	Lutjanus sebae	Called seaperch in Australia. Inhabits the areas around lagoon coral reefs and sandy bottoms.	
Wrasse	Giant Maori Wrasse	Cheilinus undulatus	This is the largest of all the wrasse. It is found in dense reef areas, feeding on a wide variety of mollusks, fishes, sea	
Angelfish	Blue Angelfish	Pomacanthus nauarchus	Habitat is around boulders, caves, coral ledges and crevices in shallow waters. Swims alone or in groups.	

FIGURE 17.4
An example of the MdDbGrid component at design time. Notice the output of the graphics and memo fields.

Although creating the output was a simple matter of adapting the code used in the record viewer component, setting the height of the grid cells ended up being a difficult problem to solve. The lines of code for that operation may be few, but they cost me hours of work!

NOTE *Unlike the generic grid used earlier, a DBGrid is a virtual view on the dataset—there is no relation between the number of rows shown on the screen and the number of rows of data in the dataset. When you scroll up and down through the data records of the dataset, you are not scrolling through the rows of the DBGrid; the rows are stationary, and the data moves from one row to the next to give the appearance of movement. For this reason, the program doesn't try to set the height of an individual row to suit its data; it sets the height of all the data rows to a multiline height value.*

This time the control doesn't have to create a custom data link, because it is deriving from a component that already has a complex connection with the data. The new class has a new property to specify the number of lines of text for each row and overrides a few virtual methods:

```
type
  TMdDbGrid = class(TDbGrid)
  private
    FLinesPerRow: Integer;
    procedure SetLinesPerRow (Value: Integer);
  protected
    procedure DrawColumnCell(const Rect: TRect; DataCol: Integer;
      Column: TColumn; State: TGridDrawState); override;
    procedure  LayoutChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
  published
    property LinesPerRow: Integer
      read FLinesPerRow write SetLinesPerRow default 1;
  end;
```

The constructor sets the default value for the `FLinesPerRow` field. Here is the set method for the property:

```
procedure TMdDbGrid.SetLinesPerRow(Value: Integer);
begin
  if Value <> FLinesPerRow then
  begin
    FLinesPerRow := Value;
    LayoutChanged;
  end;
end;
```

The side effect of changing the number of lines is a call to the `LayoutChanged` virtual method. The system calls this method frequently when one of the many output parameters changes. In the method's code, the component first calls the inherited version and then sets the height of each row. As a basis for this computation, it uses the same formula as the `TCustomDBGrid` class: The text height is calculated using the sample word *Wg* in the current font (this text is used because it includes both a full-height uppercase character and a lowercase letter with a descender). Here's the code:

```
procedure TMdDbGrid.LayOutChanged;
var
  PixelsPerRow, PixelsTitle, I: Integer;
begin
  inherited LayOutChanged;

  Canvas.Font := Font;
  PixelsPerRow := Canvas.TextHeight('Wg') + 3;
  if dgRowLines in Options then
    Inc (PixelsPerRow, GridLineWidth);

  Canvas.Font := TitleFont;
  PixelsTitle := Canvas.TextHeight('Wg') + 4;
  if dgRowLines in Options then
    Inc (PixelsTitle, GridLineWidth);

  // set number of rows
  RowCount := 1 + (Height - PixelsTitle) div (PixelsPerRow * FLinesPerRow);

  // set the height of each row
  DefaultRowHeight := PixelsPerRow * FLinesPerRow;
  RowHeights [0] := PixelsTitle;
  for I := 1 to RowCount - 1 do
    RowHeights [I] := PixelsPerRow * FLinesPerRow;

  // send a WM_SIZE message to let the base component recompute
  // the visible rows in the private UpdateRowCount method
  PostMessage (Handle, WM_SIZE, 0, MakeLong(Width, Height));
end;
```

WARNING `Font` *and* `TitleFont` *are the grid defaults, which can be overridden by properties of the individual DBGrid column objects. This component currently ignores those settings.*

The difficult part of this method was getting the final statements right. You can set the `DefaultRowHeight` property, but in that case the title row will probably be too high. First I tried setting the `DefaultRowHeight` and then the height of the first row, but this approach complicated the code used to compute the number of visible rows in the grid (the read-only `VisibleRowCount` property). If you specify the number of rows (to avoid having rows hidden beneath the lower edge of the grid), the base class keeps recomputing it. Here's the code used to draw the data, ported from the RecordView component and adapted slightly for the grid:

```
procedure TMdDbGrid.DrawColumnCell (const Rect: TRect; DataCol: Integer;
  Column: TColumn; State: TGridDrawState);
var
  Bmp: TBitmap;
  OutRect: TRect;
```

```
begin
  if FLinesPerRow = 1 then
    inherited DrawColumnCell(Rect, DataCol, Column, State)
  else
  begin
    // clear area
    Canvas.FillRect (Rect);
    // copy the rectangle
    OutRect := Rect;
    // restrict output
    InflateRect (OutRect, -2, -2);
    // output field data
    if Column.Field is TGraphicField then
    begin
      Bmp := TBitmap.Create;
      try
        Bmp.Assign (Column.Field);
        Canvas.StretchDraw (OutRect, Bmp);
      finally
        Bmp.Free;
      end;
    end
    else if Column.Field is TMemoField then
    begin
      DrawText (Canvas.Handle, PChar (Column.Field.AsString),
        Length (Column.Field.AsString), OutRect, dt_WordBreak or dt_NoPrefix)
    end
    else // draw single line vertically centered
      DrawText (Canvas.Handle, PChar (Column.Field.DisplayText),
        Length (Column.Field.DisplayText), OutRect,
        dt_vcenter or dt_SingleLine or dt_NoPrefix);
  end;
end;
```

In this code you can see that if the user displays a single line, the grid uses the standard drawing technique with no output for memo and graphic fields. However, as soon as you increase the line count, you'll see better output.

To see this code in action, run the GridDemo example. This program has two buttons you can use to increase or decrease the row height of the grid, and two more buttons to change the font. This is an important test because the height in pixels of each cell is the height of the font multiplied by the number of lines.

Building Custom Datasets

When discussing the `TDataSet` class and the alternative families of dataset components available in Delphi in Chapter 13, "Delphi's Database Architecture," I mentioned the possibility of writing a custom dataset class. Now it's time to look at an example. The reasons for writing a custom dataset relate to the fact that you won't need to deploy a database engine, but you'll still be able to take full advantage of Delphi's database architecture, including things like persistent database fields and data-aware controls.

Writing a custom dataset is one of the most complex tasks for a component developer, so this is one of the most advanced areas (as far as low-level coding practices, including tons of pointers) in the book. Moreover, Borland hasn't released any official documentation about writing custom datasets. If you are early in your experience with Delphi, you might want to skip the rest of this chapter and come back later.

`TDataSet` is an abstract class that declares several virtual abstract methods—23 in Delphi 5, now only a handful, as most have been replaced by empty virtual methods (which you still have to override). Every subclass of `TDataSet` must override all those methods.

Before discussing the development of a custom dataset, we need to explore a few technical elements of the `TDataSet` class—in particular, record buffering. The class maintains a list of buffers that store the values of different records. These buffers store the data, but they also usually store further information for the dataset to use when managing the records. These buffers don't have a predefined structure, and each custom dataset must allocate the buffers, fill them, and destroy them. The custom dataset must also copy the data from the record buffers to the various fields of the dataset, and vice versa. In other words, the custom dataset is entirely responsible for handling these buffers.

In addition to managing the data buffers, the component is responsible for navigating among the records, managing the bookmarks, defining the structure of the dataset, and creating the proper data fields. The `TDataSet` class is nothing more than a framework; you must fill it with the appropriate code. Fortunately, most of the code follows a standard structure, which the `TDataSet`-derived VCL classes use. Once you've grasped the key ideas, you'll be able to build multiple custom datasets borrowing quite a lot of code.

To simplify this type of reuse, I've collected the common features required by any custom dataset in a `TMdCustomDataSet` class. However, I'm not going to discuss the base class first and the specific implementation later, because it would be difficult to understand. Instead, I'll detail the code required by a dataset, presenting methods of the generic and specific classes at the same time, according to a logical flow.

The Definition of the Classes

The starting point, as usual, is the declaration of the two classes discussed in this section: the generic custom dataset I've written and a specific component storing data in a file stream. The declaration of these classes is available in Listing 17.2. In addition to virtual methods, the classes contain a series of protected fields used to manage the buffers, track the current position and record count, and handle many other features. You'll also notice another record declaration at the beginning: a structure used to store the extra data for every data record you place in a buffer. The dataset places this information in each record buffer, following the data.

LISTING 17.2: THE DECLARATION OF *TMDCUSTOMDATASET* AND *TMDDATASETSTREAM*

```
// in the unit MdDsCustom
type
  EMdDataSetError = class (Exception);
```

```
TMdRecInfo = record
  Bookmark: Longint;
  BookmarkFlag: TBookmarkFlag;
end;
PMdRecInfo = ^TMdRecInfo;

TMdCustomDataSet = class(TDataSet)
protected
  // status
  FIsTableOpen: Boolean;
  // record data
  FRecordSize,       // the size of the actual data
  FRecordBufferSize, // data + housekeeping (TRecInfo)
  FCurrentRecord,    // current record (0 to FRecordCount - 1)
  BofCrack,          // before the first record (crack)
  EofCrack: Integer; // after the last record (crack)
  // create, close, and so on
  procedure InternalOpen; override;
  procedure InternalClose; override;
  function IsCursorOpen: Boolean; override;
  // custom functions
  function InternalRecordCount: Integer; virtual; abstract;
  procedure InternalPreOpen; virtual;
  procedure InternalAfterOpen; virtual;
  procedure InternalLoadCurrentRecord(Buffer: PChar); virtual; abstract;
  // memory management
  function AllocRecordBuffer: PChar; override;
  procedure InternalInitRecord(Buffer: PChar); override;
  procedure FreeRecordBuffer(var Buffer: PChar); override;
  function GetRecordSize: Word; override;
  // movement and optional navigation (used by grids)
  function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):
    TGetResult; override;
  procedure InternalFirst; override;
  procedure InternalLast; override;
  function GetRecNo: Longint; override;
  function GetRecordCount: Longint; override;
  procedure SetRecNo(Value: Integer); override;
  // bookmarks
  procedure InternalGotoBookmark(Bookmark: Pointer); override;
  procedure InternalSetToRecord(Buffer: PChar); override;
  procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
  procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
  procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
  function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
  // editing (dummy versions)
  procedure InternalDelete; override;
  procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
  procedure InternalPost; override;
```

```
    procedure InternalInsert; override;
    // other
    procedure InternalHandleException; override;
  published
    // redeclared dataset properties
    property Active;
    property BeforeOpen;
    property AfterOpen;
    property BeforeClose;
    property AfterClose;
    property BeforeInsert;
    property AfterInsert;
    property BeforeEdit;
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnCalcFields;
    property OnDeleteError;
    property OnEditError;
    property OnFilterRecord;
    property OnNewRecord;
    property OnPostError;
  end;

// in the unit MdDsStream
type
  TMdDataFileHeader = record
    VersionNumber: Integer;
    RecordSize: Integer;
    RecordCount: Integer;
  end;

  TMdDataSetStream = class(TMdCustomDataSet)
  private
    procedure SetTableName(const Value: string);
  protected
    FDataFileHeader: TMdDataFileHeader;
    FDataFileHeaderSize,   // optional file header size
    FRecordCount: Integer; // current number of records
    FStream: TStream;      // the physical table
    FTableName: string;    // table path and file name
    FFieldOffset: TList;   // field offsets in the buffer
```

```
protected
  // open and close
  procedure InternalPreOpen; override;
  procedure InternalAfterOpen; override;
  procedure InternalClose; override;
  procedure InternalInitFieldDefs; override;
  // edit support
  procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
  procedure InternalPost; override;
  procedure InternalInsert; override;
  // fields
  procedure SetFieldData(Field: TField; Buffer: Pointer); override;
  // custom dataset virutal methods
  function InternalRecordCount: Integer; override;
  procedure InternalLoadCurrentRecord(Buffer: PChar); override;
public
  procedure CreateTable;
  function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
published
  property TableName: string read FTableName write SetTableName;
end;
```

When I divided the methods into sections (as you can see by looking at the source code files), I marked each one with a roman number. You'll see those numbers in a comment describing the method, so that while browsing this long listing you'll immediately know which of the four sections you are in.

Section I: Initialization, Opening, and Closing

The first methods I'll examine are responsible for initializing the dataset and for opening and closing the file stream used to store the data. In addition to initializing the component's internal data, these methods are responsible for initializing and connecting the proper `TFields` objects to the dataset component. To make this work, all you need to do is to initialize the `FieldsDef` property with the definitions of the fields for your dataset, and then call a few standard methods to generate and bind the `TField` objects. This is the general `InternalOpen` method:

```
procedure TMdCustomDataSet.InternalOpen;
begin
  InternalPreOpen; // custom method for subclasses

  // initialize the field definitions
  InternalInitFieldDefs;

  // if there are no persistent field objects, create the fields dynamically
  if DefaultFields then
    CreateFields;
  // connect the TField objects with the actual fields
  BindFields (True);

  InternalAfterOpen; // custom method for subclasses

  // sets cracks and record position and size
  BofCrack := -1;
  EofCrack := InternalRecordCount;
  FCurrentRecord := BofCrack;
  FRecordBufferSize := FRecordSize + sizeof (TMdRecInfo);
  BookmarkSize := sizeOf (Integer);

  // everything OK: table is now open
  FIsTableOpen := True;
end;
```

You'll notice that the method sets most of the local fields of the class, and also the `BookmarkSize` field of the base `TDataSet` class. Within this method, I call two custom methods I introduced in my custom dataset hierarchy: `InternalPreOpen` and `InternalAfterOpen`. The first, `InternalPreOpen`, is used for operations required at the very beginning, such as checking whether the dataset can be opened and reading the header information from the file. The code checks an internal version number for consistency with the value saved when the table is first created, as you'll see later. By raising an exception in this method, you can eventually stop the open operation.

Here is the code for the two methods in the derived stream-based dataset:

```
const
  HeaderVersion = 10;

procedure TMdDataSetStream.InternalPreOpen;
begin
  // the size of the header
  FDataFileHeaderSize := sizeOf (TMdDataFileHeader);

  // check if the file exists
  if not FileExists (FTableName) then
    raise EMdDataSetError.Create ('Open: Table file not found');

  // create a stream for the file
  FStream := TFileStream.Create (FTableName, fmOpenReadWrite);

  // initialize local data (loading the header)
  FStream.ReadBuffer (FDataFileHeader, FDataFileHeaderSize);
  if FDataFileHeader.VersionNumber <> HeaderVersion then
    raise EMdDataSetError.Create ('Illegal File Version');
  // let's read this, double check later
  FRecordCount := FDataFileHeader.RecordCount;
end;
```

```
procedure TMdDataSetStream.InternalAfterOpen;
begin
  // check the record size
  if FDataFileHeader.RecordSize <> FRecordSize then
    raise EMdDataSetError.Create ('File record size mismatch');
  // check the number of records against the file size
 if (FDataFileHeaderSize + FRecordCount * FRecordSize) <> FStream.Size then
    raise EMdDataSetError.Create ('InternalOpen: Invalid Record Size');
end;
```

The second method, `InternalAfterOpen`, is used for operations required after the field definitions have been set and is followed by code that compares the record size read from the file against the value computed in the `InternalInitFieldDefs` method. The code also checks that the number of records read from the header is compatible with the size of the file. This test can fail if the dataset wasn't closed properly: You might want to modify this code to let the dataset refresh the record size in the header anyway.

The `InternalOpen` method of the custom dataset class is specifically responsible for calling `InternalInitFieldDefs`, which determines the field definitions (at either design time or run time). For this example, I decided to base the field definitions on an external file—an INI file that provides a section for every field. Each section contains the name and data type of the field, as well as its size if it is string data. Listing 17.3 shows the `Contrib.INI` file used in the component's demo application.

LISTING 17.3: THE *CONTRIB.INI* FILE FOR THE DEMO APPLICATION

```
[Fields]
Number = 6

[Field1]
Type = ftString
Name = Name
Size = 30

[Field2]
Type = ftInteger
Name = Level

[Field3]
Type = ftDate
Name = BirthDate

[Field4]
Type = ftCurrency
Name = Stipend

[Field5]
Type = ftString
Name = Email
Size = 50
```

```
[Field6]
Type = ftBoolean
Name = Editor
```

This file, or a similar one, must use the same name as the table file and must be in the same directory. The `InternalInitFieldDefs` method (shown in Listing 17.4) will read it, using the values it finds to set up the field definitions and determine the size of each record. The method also initializes an internal `TList` object that stores the offset of every field inside the record. You use this `TList` to access fields' data within the record buffer, as you can see in the code listing.

LISTING 17.4: THE *INTERNALINITFIELDDEFS* METHOD OF THE STREAM-BASED DATASET

```
procedure TMdDataSetStream.InternalInitFieldDefs;
var
  IniFileName, FieldName: string;
  IniFile: TIniFile;
  nFields, I, TmpFieldOffset, nSize: Integer;
  FieldType: TFieldType;
begin
  FFieldOffset := TList.Create;
  FieldDefs.Clear;
  TmpFieldOffset := 0;
  IniFilename := ChangeFileExt(FTableName, '.ini');
  Inifile := TIniFile.Create (IniFilename);
  // protect INI file
  try
    nFields := IniFile.ReadInteger (' Fields', 'Number', 0);
    if nFields = 0 then
      raise EDataSetOneError.Create (' InitFieldsDefs: 0 fields?');
    for I := 1 to nFields do
    begin
      // create the field
      FieldType := TFieldType (GetEnumValue (TypeInfo (TFieldType),
        IniFile.ReadString ('Field' + IntToStr (I), 'Type', '')));
      FieldName := IniFile.ReadString ('Field' + IntToStr (I), 'Name', '');
      if FieldName = ''  then
        raise EDataSetOneError.Create (
          'InitFieldsDefs: No name for field ' + IntToStr (I));
      nSize := IniFile.ReadInteger ('Field' + IntToStr (I), 'Size', 0);
      FieldDefs.Add (FieldName, FieldType, nSize, False);
      // save offset and compute size
      FFieldOffset.Add (Pointer (TmpFieldOffset));
      case FieldType of
        ftString:                         Inc (TmpFieldOffset, nSize + 1);
        ftBoolean, ftSmallInt, ftWord:    Inc (TmpFieldOffset, 2);
```

```
          ftInteger, ftDate, ftTime:        Inc (TmpFieldOffset, 4);
          ftFloat, ftCurrency, ftDateTime:  Inc (TmpFieldOffset, 8);
        else
          raise EDataSetOneError.Create (
            'InitFieldsDefs: Unsupported field type');
        end;
      end; // for
    finally
      IniFile.Free;
    end;
    FRecordSize := TmpFieldOffset;
  end;
```

Closing the table is a matter of disconnecting the fields (using some standard calls). Each class must dispose of the data it allocated and update the file header, the first time records are added and each time the record count has changed:

```
procedure TMdCustomDataSet.InternalClose;
begin
  // disconnect field objects
  BindFields (False);
  // destroy field object (if not persistent)
  if DefaultFields then
    DestroyFields;
  // close the file
  FIsTableOpen := False;
end;

procedure TMdDataSetStream.InternalClose;
begin
  // if required, save updated header
  if (FDataFileHeader.RecordCount <> FRecordCount) or
    (FDataFileHeader.RecordSize = 0) then
  begin
    FDataFileHeader.RecordSize := FRecordSize;
    FDataFileHeader.RecordCount := FRecordCount;
    if Assigned (FStream) then
    begin
      FStream.Seek (0, soFromBeginning);
      FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
    end;
  end;
  // free the internal list field offsets and the stream
  FFieldOffset.Free;
  FStream.Free;
  inherited InternalClose;
end;
```

Another related function is used to test whether the dataset is open, something you can solve using the corresponding local field:

```
function TMdCustomDataSet.IsCursorOpen: Boolean;
begin
  Result := FIsTableOpen;
end;
```

These are the opening and closing methods you need to implement in any custom dataset. However, most of the time, you'll also add a method to create the table. In this example, the `CreateTable` method creates an empty file and inserts information in the header: a fixed version number, a dummy record size (you don't know the size until you initialize the fields), and the record count (which is zero to start):

```
procedure TMdDataSetStream.CreateTable;
begin
  CheckInactive;
  InternalInitFieldDefs;

  // create the new file
  if FileExists (FTableName) then
    raise EMdDataSetError.Create ('File ' + FTableName + ' already exists');
  FStream := TFileStream.Create (FTableName, fmCreate or fmShareExclusive);
  try
    // save the header
    FDataFileHeader.VersionNumber := HeaderVersion;
    FDataFileHeader.RecordSize := 0;   // used later
    FDataFileHeader.RecordCount := 0;  // empty
    FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
  finally
    // close the file
    FStream.Free;
  end;
end;
```

Section II: Movement and Bookmark Management

As mentioned earlier, every dataset must implement *bookmark management*, which is necessary for navigating through the dataset. Logically, a bookmark is a reference to a specific dataset record, something that uniquely identifies the record so a dataset can access it and compare it to other records. Technically, bookmarks are pointers. You can implement them as pointers to specific data structures that store record information, or you can implement them as record numbers. For simplicity, I'll use the latter approach.

Given a bookmark, you should be able to find the corresponding record; but given a record buffer, you should also be able to retrieve the corresponding bookmark. This is the reason for appending the `TMdRecInfo` structure to the record data in each record buffer. This data structure stores the bookmark for the record in the buffer, as well as some bookmark flags defined as follows:

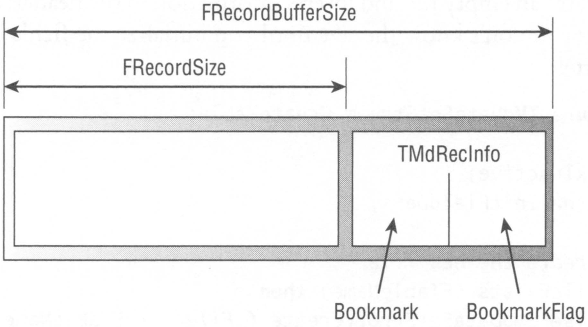
```
type
  TBookmarkFlag = (bfCurrent, bfBOF, bfEOF, bfInserted);
```

The system will request that you store these flags in each record buffer and will later ask you to retrieve the flags for a given record buffer.

To summarize, the structure of a record buffer stores the record data, the bookmark, and the bookmark flags, as you can see in Figure 17.5.

FIGURE 17.5

The structure of each buffer of the custom dataset, along with the various local fields referring to its subportions



To access the bookmark and flags, you can use as an offset the size of the data, casting the value to the `PMdRecInfo` pointer type, and then access the proper field of the `TMdRecInfo` structure via the pointer. The two methods used to set and get the bookmark flags demonstrate this technique:

```
procedure TMdCustomDataSet.SetBookmarkFlag (Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PMdRecInfo(Buffer + FRecordSize).BookmarkFlag := Value;
end;

function TMdCustomDataSet.GetBookmarkFlag (Buffer: PChar): TBookmarkFlag;
begin
  Result := PMdRecInfo(Buffer + FRecordSize).BookmarkFlag;
end;
```

The methods you use to set and get a record's current bookmark are similar to the previous two, but they add complexity because you receive a pointer to the bookmark in the `Data` parameter. Casting the value referenced by this pointer to an integer, you obtain the bookmark value:

```
procedure TMdCustomDataSet.GetBookmarkData (Buffer: PChar; Data: Pointer);
begin
  Integer(Data^) := PMdRecInfo(Buffer + FRecordSize).Bookmark;
end;

procedure TMdCustomDataSet.SetBookmarkData (Buffer: PChar; Data: Pointer);
```

```
begin
  PMdRecInfo(Buffer + FRecordSize).Bookmark := Integer(Data^);
end;
```

The key bookmark management method is `InternalGotoBookmark`, which your dataset uses to make a given record the current one. This isn't the standard navigation technique—it's much more common to move to the next or previous record (something you can accomplish using the `GetRecord` method presented in the next section), or to move to the first or last record (something you'll accomplish using the `InternalFirst` and `InternalLast` methods described shortly).

Oddly enough, the `InternalGotoBookmark` method doesn't expect a bookmark parameter, but a pointer to a bookmark, so you must dereference it to determine the bookmark value. You use the following method, `InternalSetToRecord`, to jump to a given bookmark, but it must extract the bookmark from the record buffer passed as a parameter. Then, `InternalSetToRecord` calls `InternalGotoBookmark`. Here are the two methods:

```
procedure TMdCustomDataSet.InternalGotoBookmark (Bookmark: Pointer);
var
  ReqBookmark: Integer;
begin
  ReqBookmark := Integer (Bookmark^);
  if (ReqBookmark >= 0) and (ReqBookmark < InternalRecordCount) then
    FCurrentRecord := ReqBookmark
  else
    raise EMdDataSetError.Create ('Bookmark ' +
      IntToStr (ReqBookmark) + ' not found');
end;

procedure TMdCustomDataSet.InternalSetToRecord (Buffer: PChar);
var
  ReqBookmark: Integer;
begin
  ReqBookmark := PMdRecInfo(Buffer + FRecordSize).Bookmark;
  InternalGotoBookmark (@ReqBookmark);
end;
```

In addition to the bookmark management methods just described, you use several other navigation methods to move to specific positions within the dataset, such as the first or last record. These two methods don't really move the current record pointer to the first or last record, but move it to one of two special locations before the first record and after the last one. These are not actual records: Borland calls them *cracks*. The beginning-of-file crack, or `BofCrack`, has the value –1 (set in the `InternalOpen` method), because the position of the first record is zero. The end-of-file crack, or `EofCrack`, has the value of the number of records, because the last record has the position `FRecordCount - 1`. I used two local fields, called `EofCrack` and `BofCrack`, to make this code easier to read:

```
procedure TMdCustomDataSet.InternalFirst;
begin
  FCurrentRecord := BofCrack;
end;
```

```
procedure TMdCustomDataSet.InternalLast;
begin
  EofCrack := InternalRecordCount;
  FCurrentRecord := EofCrack;
end;
```

The `InternalRecordCount` method is a virtual method introduced in my `TMdCustomDataSet` class, because different datasets can either have a local field for this value (as in case of the stream-based dataset, which has an `FRecordCount` field) or compute it on the fly.

Another group of optional methods is used to get the current record number (used by the DBGrid component to show a proportional vertical scroll bar), set the current record number, or determine the number of records. These methods are easy to understand, if you recall that the range of the internal `FCurrentRecord` field is from 0 to the number of records minus 1. In contrast, the record number reported to the system ranges from 1 to the number of records:

```
function TMdCustomDataSet.GetRecordCount: Longint;
begin
  CheckActive;
  Result := InternalRecordCount;
end;

function TMdCustomDataSet.GetRecNo: Longint;
begin
  UpdateCursorPos;
  if FCurrentRecord < 0 then
    Result := 1
  else
    Result := FCurrentRecord + 1;
end;

procedure TMdCustomDataSet.SetRecNo(Value: Integer);
begin
  CheckBrowseMode;
  if (Value > 1) and (Value <= FRecordCount) then
  begin
    FCurrentRecord := Value - 1;
    Resync([]);
  end;
end;
```

Notice that the generic custom dataset class implements all the methods of this section. The derived stream-based dataset doesn't need to modify any of them.

Section III: Record Buffers and Field Management

Now that we've covered all the support methods, let's examine the core of a custom dataset. In addition to opening and creating records and moving around between them, the component needs to move the data from the stream (the persistent file) to the record buffers, and from the record buffers to the `TField` objects that are connected to the data-aware controls. The management of record buffers is complex, because each dataset also needs to allocate, empty, and free the memory it requires:

```
function TMdCustomDataSet.AllocRecordBuffer: PChar;
begin
  GetMem (Result, FRecordBufferSize);
end;

procedure TMdCustomDataSet.FreeRecordBuffer (var Buffer: PChar);
begin
  FreeMem (Buffer);
end;
```

You allocate memory this way because a dataset generally adds more information to the record buffer, so the system has no way of knowing how much memory to allocate. Notice that in the `AllocRecordBuffer` method, the component allocates the memory for the record buffer, including both the database data and the record information. In the `InternalOpen` method, I wrote the following:

```
FRecordBufferSize := InternalRecordSize + sizeof (TMdRecInfo);
```

The component also needs to implement a function to reset the buffer (`InternalInitRecord`), usually filling it with numeric zeros or spaces.

Oddly enough, you must also implement a method that returns the size of each record, but only the data portion—not the entire record buffer. This method is necessary for implementing the read-only `RecordSize` property, which is used only in a couple of peculiar cases in the entire VCL source code. In the generic custom dataset, the `GetRecordSize` method returns the value of the `FRecordSize` field.

Now we've reached the core of the custom dataset component. The methods in this group are `GetRecord`, which reads data from the file; `InternalPost` and `InternalAddRecord`, which update or add new data to the file; and `InternalDelete`, which removes data and is not implemented in the sample dataset.

The most complex method of this group is `GetRecord`, which serves multiple purposes. The system uses this method to retrieve the data for the current record, fill a buffer passed as a parameter, and retrieve the data of the next or previous records. The `GetMode` parameter determines its action:

```
type
  TGetMode = (gmCurrent, gmNext, gmPrior);
```

Of course, a previous or next record might not exist. Even the current record might not exist—for example, when the table is empty (or in case of an internal error). In these cases you don't retrieve the data but return an error code. Therefore, this method's result can be one of the following values:

```
type
  TGetResult = (grOK, grBOF, grEOF, grError);
```

Checking to see if the requested record exists is slightly different than you might expect. You don't have to determine if the current record is in the proper range, only if the requested record is. For example, in the `gmCurrent` branch of the `case` statement, you use the standard expression `CurrentRecord>= InternalRecourdCount`. To fully understand the various cases, you might want to read the code a couple of times.

It took me some trial and error (and system crashes caused by recursive calls) to get the code straight when I wrote my first custom dataset a few years back. To test it, consider that if you use a DBGrid, the system will perform a series of `GetRecord` calls, until either the grid is full or `GetRecord` return grEOF. Here's the entire code for the `GetRecord` method:

```
// III: Retrieve data for current, previous, or next record
// (moving to it if necessary) and return the status
function TMdCustomDataSet.GetRecord(Buffer: PChar;
  GetMode: TGetMode; DoCheck: Boolean): TGetResult;
begin
  Result := grOK; // default
  case GetMode of
    gmNext: // move on
      if FCurrentRecord < InternalRecordCount - 1 then
        Inc (FCurrentRecord)
      else
        Result := grEOF; // end of file
    gmPrior: // move back
      if FCurrentRecord > 0 then
        Dec (FCurrentRecord)
      else
        Result := grBOF; // begin of file
    gmCurrent: // check if empty
      if FCurrentRecord >= InternalRecordCount then
        Result := grError;
  end;
  // load the data
  if Result = grOK then
    InternalLoadCurrentRecord (Buffer)
  else if (Result = grError) and DoCheck then
    raise EMdDataSetError.Create ('GetRecord: Invalid record');
end;
```

If there's an error and the `DoCheck` parameter was True, `GetRecord` raises an exception. If everything goes fine during record selection, the component loads the data from the stream, moving to the position of the current record (given by the record size multiplied by the record number). In addition, you need to initialize the buffer with the proper bookmark flag and bookmark (or record number) value. This is accomplished by another virtual method I introduced, so that derived classes will only need to implement this portion of the code, while the complex `GetRecord` method remains unchanged:

```
procedure TMdDataSetStream.InternalLoadCurrentRecord (Buffer: PChar);
begin
  FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
  FStream.ReadBuffer (Buffer^, FRecordSize);
  with PMdRecInfo(Buffer + FRecordSize)^ do
  begin
    BookmarkFlag := bfCurrent;
```

```
      Bookmark := FCurrentRecord;
    end;
  end;
```

You move data to the file in two different cases: when you modify the current record (that is, a post after an edit) or when you add a new record (a post after an insert or append). You use the `InternalPost` method in both cases, but you can check the dataset's `State` property to determine which type of post you're performing. In both cases, you don't receive a record buffer as a parameter; so, you must use the `ActiveRecord` property of `TDataSet`, which points to the buffer for the current record:

```
procedure TMdDataSetStream.InternalPost;
begin
  CheckActive;
  if State = dsEdit then
  begin
    // replace data with new data
    FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
  end
  else
  begin
    // always append
    InternalLast;
    FStream.Seek (0, soFromEnd);
    FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
    Inc (FRecordCount);
  end;
end;
```

In addition, there's another related method: `InternalAddRecord`. This method is called by the `AddRecord` method, which in turn is called by `InsertRecord` and `AppendRecord`. These last two are public methods a user can call. This is an alternative to inserting or appending a new record to the dataset, editing the values of the various fields, and then posting the data, because the `InsertRecord` and `AppendRecord` calls receive the values of the fields as parameters. All you must do at that point is replicate the code used to add a new record in the `InternalPost` method:

```
procedure TMdDataSetOne.InternalAddRecord(Buffer: Pointer; Append: Boolean);
begin
  // always append at the end
  InternalLast;
  FStream.Seek (0, soFromEnd);
  FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
  Inc (FRecordCount);
end;
```

I should also have implemented a file operation that removes the current record. This operation is common, but it is complex. If you take a simple approach, such as creating an empty spot in the file, then you'll need to keep track of that spot and make the code for reading or writing a specific record work around that location. An alternate solution is to make a copy of the entire file without the given record and then replace the original file with the copy. Given these choices, I felt that for this example I could forgo supporting record deletion.

Section IV: From Buffers to Fields

In the last few methods, you've seen how datasets move data from the data file to the memory buffer. However, there's little Delphi can do with this record buffer, because it doesn't yet know how to interpret the data in the buffer. You need to provide two more methods: `GetData`, which copies the data from the record buffer to the field objects of the dataset, and `SetData`, which moves the data back from the fields to the record buffer. Delphi will automatically move the data from the field objects to the data-aware controls and back.

The code for these two methods isn't difficult, primarily because you saved the field offsets inside the record data in a `TList` object called `FFieldOffset`. By incrementing the pointer to the initial position in the record buffer of the current field's offset, you can get the specific data, which takes `Field.DataSize` bytes.

A confusing element of these two methods is that they both accept a `Field` parameter and a `Buffer` parameter. At first, you might think the buffer passed as a parameter is the record buffer. However, I found out that the `Buffer` is a pointer to the field object's raw data. If you use one of the field object's methods to move that data, it will call the dataset's `GetData` or `SetData` method, probably causing an infinite recursion. Instead, you should use the `ActiveBuffer` pointer to access the record buffer, use the proper offset to get to the data for the current field in the record buffer, and then use the provided `Buffer` to access the field data. The only difference between the two methods is the direction you move the data:

```
function TMdDataSetOne.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FieldOffset: Integer;
  Ptr: PChar;
begin
  Result := False;
  if not IsEmpty and (Field.FieldNo > 0) then
  begin
    FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
    Ptr := ActiveBuffer;
    Inc (Ptr, FieldOffset);
    if Assigned (Buffer) then
      Move (Ptr^, Buffer^, Field.DataSize);
    Result := True;
    if (Field is TDateTimeField) and (Integer(Ptr^) = 0) then
      Result := False;
  end;
end;

procedure TMdDataSetOne.SetFieldData(Field: TField; Buffer: Pointer);
var
  FieldOffset: Integer;
  Ptr: PChar;
```

```
begin
  if Field.FieldNo >= 0 then
  begin
    FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
    Ptr := ActiveBuffer;
    Inc (Ptr, FieldOffset);
    if Assigned (Buffer) then
      Move (Buffer^, Ptr^, Field.DataSize)
    else
      raise Exception.Create (
        'Very bad error in TMdDataSetStream.SetField data');
    DataEvent (deFieldChange, Longint(Field));
  end;
end;
```

The `GetField` method should return True or False to indicate whether the field contains data or is empty (a null field, to be more precise). However, unless you use a special marker for blank fields, it's difficult to determine this condition, because you're storing values of different data types. For example, a test such as `Ptr^<>#0` makes sense only if you are using a string representation for all the fields. If you use this test, zero integer values and empty strings will show as null values (the data-aware controls will be empty), which may be what you want. The problem is that Boolean False values won't show up. Even worse, floating-point values with no decimals and few digits won't be displayed, because the exponent portion of their representation will be zero. However, to make this example work, I had to consider as empty each date/time field with an initial zero. Without this code, Delphi tries to convert the illegal *internal* zero date (internally, date fields don't use a `TDateTime` data type but a different representation), raising an exception. The code used to work with past versions of Delphi.

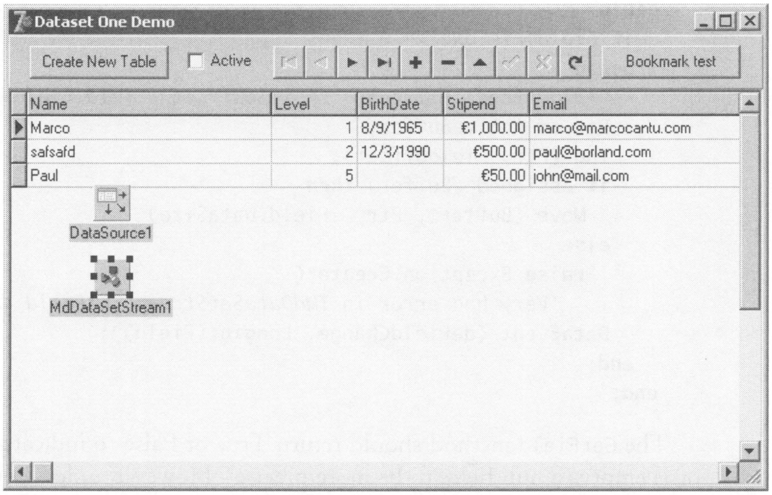
WARNING *While trying to fix this problem, I also found out that if you call* `IsNull` *for a field, this request is resolved by calling* `GetFieldData` *without passing any buffer to fill but looking only for the result of the function call. This is the reason for the* `if Assigned (Buffer)` *test within the code.*

There's one final method, which doesn't fall into any category: `InternalHandleException`. Generally, this method silences the exception, because it is activated only at design time.

Testing the Stream-Based DataSet

After all this work, you're ready to test an application example of the custom dataset component, which is installed in the component's package for this chapter. The form displayed by the StreamDS-Demo program is simple, as you can see in Figure 17.6. It has a panel with two buttons, a check box, and a navigator component, plus a DBGrid filling its client area.

Figure 17.6 shows the example's form at design time, but I activated the custom dataset so that its data is visible. I already prepared the INI file with the table definition (the file listed earlier when discussing the dataset initialization), and I executed the program to add some data to the file.



Name	Level	BirthDate	Stipend	Email
Marco	1	8/9/1965	€1,000.00	marco@marcocantu.com
safsafd	2	12/3/1990	€500.00	paul@borland.com
Paul	5		€50.00	john@mail.com

FIGURE 17.6 The form of the StreamDSDemo example. The custom dataset has been activated, so you can already see the data at design time.

You can also modify the form using Delphi's Fields editor and set the properties of the various field objects. Everything works as it does with one of the standard dataset controls. However, to make this work, you must enter the name of the custom dataset's file in the `TableName` property, using the complete path.

WARNING *The demo program defines the absolute path of the table file at design time, so you'll need to fix it if you copy the examples to a different drive or directory. In the example, the* `TableName` *property is used only at design time. At run time, the program looks for the table in the current directory.*

The example code is simple, especially compared to the custom dataset code. If the table doesn't exist yet, you can click the Create New Table button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MdDataSetStream1.CreateTable;
  MdDataSetStream1.Open;
  CheckBox1.Checked := MdDataSetStream1.Active;
end;
```

You create the file first, opening and closing it within the `CreateTable` call, and then open the table. This is the same behavior as the `TTable` component (which accomplishes this step using the `CreateTable` method). To open or close the table, you can click the check box:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  MdDataSetStream1.Active := CheckBox1.Checked;
end;
```

Finally, I created a method that tests the custom dataset's bookmark management code (it works).

A Directory in a Dataset

An important idea related to datasets in Delphi is that they represent a set of data, regardless of where this data comes from. A SQL server and a local file are examples of traditional datasets, but you can use the same technology to show a list of a system's users, a list of a folder's files, the properties of objects, XML-based data, and so on.

As an example, the second dataset presented in this chapter is a list of files. I've built a generic dataset based on a list of objects in memory (using a `TObjectList`), and then derived a version in which the objects correspond to a folder's files. The example is simplified by the fact that it is a read-only dataset, so you might find it more straightforward than the previous dataset.

NOTE *Some of the ideas presented here were discussed in an article I wrote for the Borland Community website, published in June 2000 at the URL* `bdn.borland.com/article/0,1410,20587,00.html`.

A List as a Dataset

The generic list-based dataset is called `TMdListDataSet` and contains the list of objects, a list that is created when you open the dataset and freed when you close it. This dataset doesn't store the record data within the buffer; rather, it saves in the buffer only the position in the list of the entry corresponding to the record's data. This is the class definition:

```
type
  TMdListDataSet = class (TMdCustomDataSet)
  protected
    // the list holding the data
    FList: TObjectList;
    // dataset virtual methods
    procedure InternalPreOpen; override;
    procedure InternalClose; override;
    // custom dataset virtual methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord (Buffer: PChar); override;
  end;
```

You can see that by writing a generic custom data class, you can override a few virtual methods of the `TDataSet` class and of this custom dataset class, and have a working dataset (although this is still an abstract class, which requires extra code from subclasses to work). When the dataset is opened, you have to create the list and set the record size, to indicate you're saving the list index in the buffer:

```
procedure TMdListDataSet.InternalPreOpen;
begin
  FList := TObjectList.Create (True); // owns the objects
  FRecordSize := 4; // an integer, the list item id
end;
```

Further derived classes at this point should also fill the list with objects.

TIP *Like the ClientDataSet, my list dataset keeps its data in memory. However, using some smart techniques, you can also create a list of fake objects and then load the actual objects only when you are accessing them.*

Closing is a matter of freeing the list, which has a record count corresponding to the list size:

```
function TMdListDataSet.InternalRecordCount: Integer;
begin
  Result := fList.Count;
end;
```

The only other method saves the current record's data in the record buffer, including the bookmark information. The core data is the position of the current record, which matches the list index (and also the bookmark):

```
procedure TMdListDataSet.InternalLoadCurrentRecord (Buffer: PChar);
begin
  PInteger (Buffer)^ := fCurrentRecord;
  with PMdRecInfo(Buffer + FRecordSize)^ do
  begin
    BookmarkFlag := bfCurrent;
    Bookmark := fCurrentRecord;
  end;
end;
```

Directory Data

The derived directory dataset class has to provide a way to load the objects in memory when the dataset is opened, to define the proper fields, and to read and write the value of those fields. It also has a property indicating the directory to work on—or, to be more precise, the directory plus the file mask used for filtering the files (such as `c:\docs\*.txt`):

```
type
  TMdDirDataset = class(TMdListDataSet)
  private
    FDirectory: string;
    procedure SetDirectory(const NewDirectory: string);
  protected
    // TDataSet virtual methods
    procedure InternalInitFieldDefs; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    function GetCanModify: Boolean; override;
    // custom dataset virtual methods
    procedure InternalAfterOpen; override;
  public
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
  published
    property Directory: string read FDirectory write SetDirectory;
  end;
```

The `GetCanModify` function is another virtual method of `TDataSet`, used to determine if the dataset is read-only. In this case, it returns False. You don't have to write any code for the `SetFieldData` procedure, but you must define it because it is an abstract virtual method.

Because you are dealing with a list of objects, the unit includes a class for those objects. In this case, the file data is extracted from a `TSearchRec` buffer by the `TFileData` class constructor:

```
type
  TFileData = class
  public
    ShortFileName: string;
    Time: TDateTime;
    Size: Integer;
    Attr: Integer;
    constructor Create (var FileInfo: TSearchRec);
  end;

constructor TFileData.Create (var FileInfo: TSearchRec);
begin
  ShortFileName := FileInfo.Name;
  Time := FileDateToDateTime (FileInfo.Time);
  Size := FileInfo.Size;
  Attr := FileInfo.Attr;
end;
```

This constructor is called for each folder while opening the dataset:

```
procedure TMdDirDataset.InternalAfterOpen;
var
  Attr: Integer;
  FileInfo: TSearchRec;
  FileData: TFileData;
begin
  // scan all files
  Attr := faAnyFile;
  FList.Clear;
  if SysUtils.FindFirst(fDirectory, Attr, FileInfo) = 0 then
  repeat
    FileData := TFileData.Create (FileInfo);
    FList.Add (FileData);
  until SysUtils.FindNext(FileInfo) <> 0;
  SysUtils.FindClose(FileInfo);
end;
```

The next step is to define the fields of the dataset, which in this case are fixed and depend on the available directory data:

```
procedure TMdDirDataset.InternalInitFieldDefs;
begin
  if fDirectory = '' then
    raise EMdDataSetError.Create ('Missing directory');

  // field definitions
  FieldDefs.Clear;
  FieldDefs.Add ('FileName', ftString, 40, True);
  FieldDefs.Add ('TimeStamp', ftDateTime);
  FieldDefs.Add ('Size', ftInteger);
  FieldDefs.Add ('Attributes', ftString, 3);
  FieldDefs.Add ('Folder', ftBoolean);
end;
```

Finally, the component has to move the data from the list object referenced by the current record buffer (the `ActiveBuffer` value) to each field of the dataset, as requested by the `GetFieldData` method. This function uses either `Move` or `StrCopy`, depending on the data type, and it does some conversions for the attribute codes (*H* for hidden, *R* for read-only, and *S* for system) extracted from the related flags and used to determine whether a file is a folder. Here is the code:

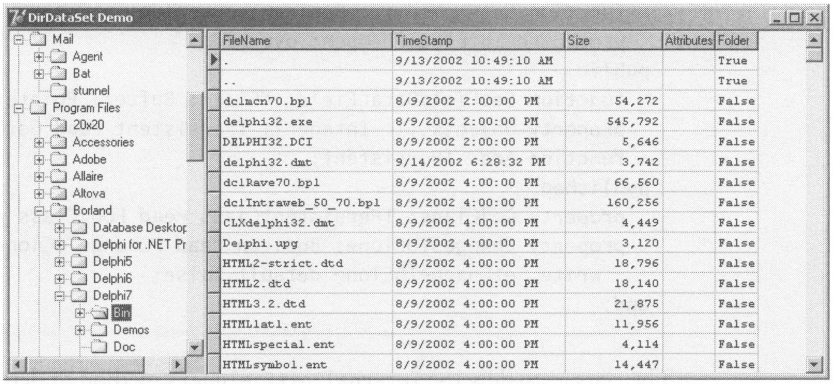
```
function TMdDirDataset.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FileData: TFileData;
  Bool1: WordBool;
  strAttr: string;
  t: TDateTimeRec;
begin
  FileData := fList [Integer(ActiveBuffer^)] as TFileData;
  case Field.Index of
    0: // filename
      StrCopy (Buffer, pchar(FileData.ShortFileName));
    1: // timestamp
    begin
      t := DateTimeToNative (ftdatetime, FileData.Time);
      Move (t, Buffer^, sizeof (TDateTime));
    end;
    2: // size
      Move (FileData.Size, Buffer^, sizeof (Integer));
    3: // attributes
    begin
      strAttr := '   ';
      if (FileData.Attr and SysUtils.faReadOnly) > 0 then
        strAttr [1] := 'R';
      if (FileData.Attr and SysUtils.faSysFile) > 0 then
        strAttr [2] := 'S';
      if (FileData.Attr and SysUtils.faHidden) > 0 then
        strAttr [3] := 'H';
      StrCopy (Buffer, pchar(strAttr));
    end;
    4: // folder
    begin
      Bool1 := FileData.Attr and SysUtils.faDirectory > 0;
      Move (Bool1, Buffer^, sizeof (WordBool));
    end;
  end; // case
  Result := True;
end;
```

The tricky part in writing this code was figuring out the internal format of dates stored in date/time fields. This is not the common `TDateTime` format used by Delphi, and not even the internal `TTimeStamp`, but what is internally called the *native date and time format*. I've written a conversion function by cloning one I found in the VCL code for the date/time fields:

```
function DateTimeToNative(DataType: TFieldType; Data: TDateTime): TDateTimeRec;
var
  TimeStamp: TTimeStamp;
begin
  TimeStamp := DateTimeToTimeStamp(Data);
  case DataType of
    ftDate: Result.Date := TimeStamp.Date;
    ftTime: Result.Time := TimeStamp.Time;
  else
    Result.DateTime := TimeStampToMSecs(TimeStamp);
  end;
end;
```

With this dataset available, building the demo program (shown in Figure 17.7) was just a matter of connecting a DBGrid component to the dataset and adding a folder-selection component, the sample ShellTreeView control. This control is set up to work only on files, by setting its `Root` property to C:\. When the user selects a new folder, the `OnChange` event handler of the ShellTreeView control refreshes the dataset:

```
procedure TForm1.ShellTreeView1Change(Sender: TObject; Node: TTreeNode);
begin
  MdDirDataset1.Close;
  MdDirDataset1.Directory := ShellTreeView1.Path + '\*.*';
  MdDirDataset1.Open;
end;
```



FileName	TimeStamp	Size	Attributes	Folder
.	9/13/2002 10:49:10 AM			True
..	9/13/2002 10:49:10 AM			True
dclmcn70.bpl	8/9/2002 2:00:00 PM	54,272		False
delphi32.exe	8/9/2002 2:00:00 PM	545,792		False
DELPHI32.DCI	8/9/2002 2:00:00 PM	5,646		False
delphi32.dmt	9/14/2002 6:28:32 PM	3,742		False
dclRave70.bpl	8/9/2002 4:00:00 PM	66,560		False
dclIntraweb_50_70.bpl	8/9/2002 4:00:00 PM	160,256		False
CLXdelphi32.dmt	8/9/2002 4:00:00 PM	4,449		False
Delphi.upg	8/9/2002 4:00:00 PM	3,120		False
HTML2-strict.dtd	8/9/2002 4:00:00 PM	18,796		False
HTML2.dtd	8/9/2002 4:00:00 PM	18,140		False
HTML3.2.dtd	8/9/2002 4:00:00 PM	21,875		False
HTMLlat1.ent	8/9/2002 4:00:00 PM	11,956		False
HTMLspecial.ent	8/9/2002 4:00:00 PM	4,114		False
HTMLsymbol.ent	8/9/2002 4:00:00 PM	14,447		False

FIGURE 17.7

The output of the DirDemo example, which uses an unusual dataset that shows directory data

WARNING *If your version of Windows has problems with the sample shell controls available in Delphi, you can use the DirDemoNoShell version of the example, which uses the old-fashioned Windows 3.1–compatible Delphi file controls.*

A Dataset of Objects

As you saw in the previous example, a list of objects is conceptually similar to the rows of a table in a dataset. In Delphi, you can build a dataset wrapping a list of objects, as in the case of the `TFileData` class. It's intriguing to extend this example to build a dataset that supports generic objects, which you can do thanks to the extended RTTI available in Delphi.

This dataset component inherits from `TMdListDataSet`, as in the previous example. You must provide a single setting: the target class, stored in the `ObjClass` property (see the complete definition of the `TMdObjDataSet` class in Listing 17.5).

LISTING 17.5: THE COMPLETE DEFINITION OF THE *TMDOBJDATASET* CLASS

```
type
 TMdObjDataSet = class(TMdListDataSet)
 private
   PropList: PPropList;
   nProps: Integer;
   FObjClass: TPersistentClass;
   ObjClone: TPersistent;
   FChangeToClone: Boolean;
   procedure SetObjClass (const Value: TPersistentClass);
   function GetObjects (I: Integer): TPersistent;
   procedure SetChangeToClone (const Value: Boolean);
 protected
   procedure InternalInitFieldDefs; override;
   procedure InternalClose; override;
   procedure InternalInsert; override;
   procedure InternalPost; override;
   procedure InternalCancel; override;
   procedure InternalEdit; override;
   procedure SetFieldData(Field: TField; Buffer: Pointer); override;
   function GetCanModify: Boolean; override;
   procedure InternalPreOpen; override;
 public
   function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
   property Objects [I: Integer]: TPersistent read GetObjects;
   function Add: TPersistent;
 published
   property ObjClass: TPersistentClass read FObjClass write SetObjClass;
   property ChangesToClone: Boolean read FChangeToClone
     write SetChangeToClone default False;
 end;
```

The class is used by the `InternalInitFieldDefs` method to determine the dataset fields based on the published properties of the target class, which are extracted using RTTI:

```
procedure TMdObjDataSet.InternalInitFieldDefs;
var
  i: Integer;
```

```
begin
  if FObjClass = nil then
    raise Exception.Create ('TMdObjDataSet: Unassigned class');

  // field definitions
  FieldDefs.Clear;
  nProps := GetTypeData(fObjClass.ClassInfo)^.PropCount;
  GetMem(PropList, nProps * SizeOf(Pointer));
  GetPropInfos (fObjClass.ClassInfo, PropList);

  for i := 0 to nProps - 1 do
    case PropList [i].PropType^.Kind of
      tkInteger, tkEnumeration, tkSet:
        FieldDefs.Add (PropList [i].Name, ftInteger, 0);
      tkChar: FieldDefs.Add (PropList [i].Name, ftFixedChar, 0);
      tkFloat: FieldDefs.Add (PropList [i].Name, ftFloat, 0);
      tkString, tkLString:
        FieldDefs.Add (PropList [i].Name, ftString, 50); // TODO: fix size
      tkWString: FieldDefs.Add (PropList [i].Name, ftWideString, 50);
        // TODO: fix size
    end;
end;
```

Similar RTTI-based code is used in the `GetFieldData` and `SetFieldData` methods to access the properties of the current object when a dataset field access operation is requested. The huge advantage in using properties to access the dataset data is that read and write operations can be mapped directly to data but also use the corresponding method. This way, you can write the *business rules* of your application by implementing rules in the read and write methods of the properties—definitely a sounder OOP approach than hooking code to field objects and validating them.

Here is a slightly simplified version of GetFieldData (the other method is symmetric):

```
function TObjDataSet.GetFieldData (
  Field: TField; Buffer: Pointer): Boolean;
var
  Obj: TPersistent;
  TypeInfo: PTypeInfo;
  IntValue: Integer;
  FlValue: Double;
begin
  if FList.Count = 0 then
  begin
    Result := False;
    exit;
  end;
  Obj := fList [Integer(ActiveBuffer^)] as TPersistent;
  TypeInfo := PropList [Field.FieldNo-1]^.PropType^;
  case TypeInfo.Kind of
    tkInteger, tkChar, tkWChar, tkClass, tkEnumeration, tkSet:
        begin
          IntValue := GetOrdProp(Obj, PropList [Field.FieldNo-1]);
          Move (IntValue, Buffer^, sizeof (Integer));
        end;
      tkFloat:
        begin
          FlValue := GetFloatProp(Obj, PropList [Field.FieldNo-1]);
          Move (FlValue, Buffer^, sizeof (Double));
        end;
      tkString, tkLString, tkWString:
        StrCopy (Buffer, pchar(GetStrProp(Obj, PropList [Field.FieldNo-1])));
    end;
    Result := True;
  end;
```

This pointer-based code may look terrible, but if you've endured the discussion of the technical details of developing a custom dataset, it won't add much complexity to the picture. It uses some of the data structures defined (and briefly commented) in the TypInfo unit, which should be your reference for any questions about the previous code.

Using this naïve approach of editing the object's data directly, you might wonder what happens if a user cancels the editing operation (something Delphi generally accounts for). My dataset provides two alternative approaches, controlled by the `ChangesToClone` property and based on the idea of cloning objects by copying their published properties. The core `DoClone` procedure uses RTTI code similar to what you have already seen to copy all the published data of an object into another object, creating an effective copy (or a clone).

This cloning takes place in both cases. Depending on the value of the `ChangesToClone` property, either the edit operations are performed on the clone object, which is then copied over the actual object during the Post operation; or the edit operations are performed on the actual object, and the clone is used to get back the original values if editing terminates with a Cancel request. This is the code of the three methods involved:

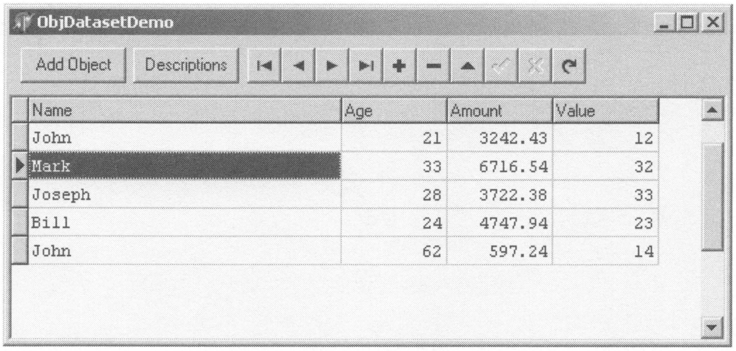
```
procedure TObjDataSet.InternalEdit;
begin
  DoClone (fList [FCurrentRecord] as TDbPers, ObjClone);
end;

procedure TObjDataSet.InternalPost;
begin
  if FChangeToClone and Assigned (ObjClone) then
    DoClone (ObjClone, TDbPers (fList [fCurrentRecord]));
end;

procedure TMdObjDataSet.InternalCancel;
begin
  if not FChangeToClone and Assigned (ObjClone) then
    DoClone (ObjClone, TPersistent(fList [fCurrentRecord]));
end;
```

In the `SetFieldData` method, you have to modify either the cloned object or the original instance. To make things more complicated, you must also consider this difference in the `GetFieldData` method: If you are reading fields from the current object, you might have to use its modified clone (otherwise, the user's changes to other fields will disappear).

As you can see in Listing 17.5, the class also has an `Objects` array that accesses the data in an OOP way and an `Add` method that's similar to the `Add` method of a collection. By calling `Add`, the code creates a new empty object of the target class and adds it to the internal list:

```
function TMdObjDataSet.Add: TPersistent;
begin
  if not Active then
    Open;
  Result := fObjClass.Create;
  fList.Add (Result);
end;
```

To demonstrate the use of this component, I wrote the ObjDataSetDemo example. It has a demo target class with a few fields and buttons to create objects automatically, as you can see in Figure 17.8. The most interesting feature of the program, however, is something you have to try for yourself. Run the program and look at the DbGrid columns. Then edit the target class, `TDemo`, adding a new published property to it. Run the program again, and the grid will include a new column for the property.



Name	Age	Amount	Value
John	21	3242.43	12
Mark	33	6716.54	32
Joseph	28	3722.38	33
Bill	24	4747.94	23
John	62	597.24	14

FIGURE 17.8
The ObjDataSetDemo example showcases a dataset mapped to objects using RTTI.

What's Next?

In this chapter we've delved into Delphi's database architecture by first examining the development of data-aware controls and then studying the internals of the `TDataSet` class to write a couple of custom dataset components. With this information and the other ideas presented in this part of the book, you should be able to choose the architecture of your database applications, depending on your needs.

Database programming is a core element of Delphi, which is why I've devoted several chapters to this topic. I'll continue to cover it in the next chapter, which is devoted to the new reporting engine available in Delphi 7. We'll get back to databases when focusing on presenting data over the Web in Chapters 20 and 21, and also when discussing XML and SOAP in Chapters 22 and 23.

Chapter 18

Reporting with Rave

DATABASE APPLICATIONS LET YOU view and edit data, but quite often their output should be physically printed to paper. Technically, Delphi supports printing in many different ways, from direct text output to the use of the printer `Canvas`, from sophisticated database reporting to the generation of documents in various formats (from Microsoft's Word to Sun's OpenOffice). In this chapter we'll focus exclusively on reports and in particular on the use of the third-party reporting Rave engine included in Delphi 7. If you are interested in other Delphi techniques for driving the printer, see the related material on my website (discussed in Appendix C, "Free Companion Books on Delphi").

Reporting tools are important because they can perform complex processing by themselves: the reporting subsystem can become a stand-alone application. Although this chapter focuses on how you can produce a report from the dataset within your Delphi programs, you should always keep in mind the autonomous nature of reports evaluating such a tool.

Reports must be created with care, because they represent a user interface for your applications that goes beyond and is sometimes more significant than the software itself. Many more people will probably look at the printed reports than just the users who produce the reports using the programs. For this reason, it's important to have good-quality reports and a flexible architecture to let users customize them.

NOTE *This chapter was written with the help of Jim Gunkel of Nevrona Designs, the company that developed the Rave engine.*

This chapter covers the following topics:

- Introducing Rave (Report Authoring Visual Environment)
- Rave Delphi components
- Rave Designer components
- From databases to reports
- Advanced features of Rave

Introducing Rave

Reports are a primary means of retrieving information from the data being managed by an application. To solve the problems associated with presenting a visual report of data in a meaningful and informative manner, traditional visual reporting applications have offered banded layout tools geared toward table-style data listings. Today, however, more complex reporting requirements exist that are not easily handled by banded layout tools.

Rave Reports is a visual report design environment offering many unique features that help make the reporting process simpler, quicker, and more efficient. Rave can handle a wide variety of report formats and includes advanced technologies such as mirroring, to encourage the reuse of report contents for quicker changes and easier maintenance.

This chapter presents a brief introduction to Rave's features. Additional information about Rave can be found in the online Help files, in the PDF documentation on Delphi's CD, in several demo projects, and on the producer's website, `www.nevrona.com`.

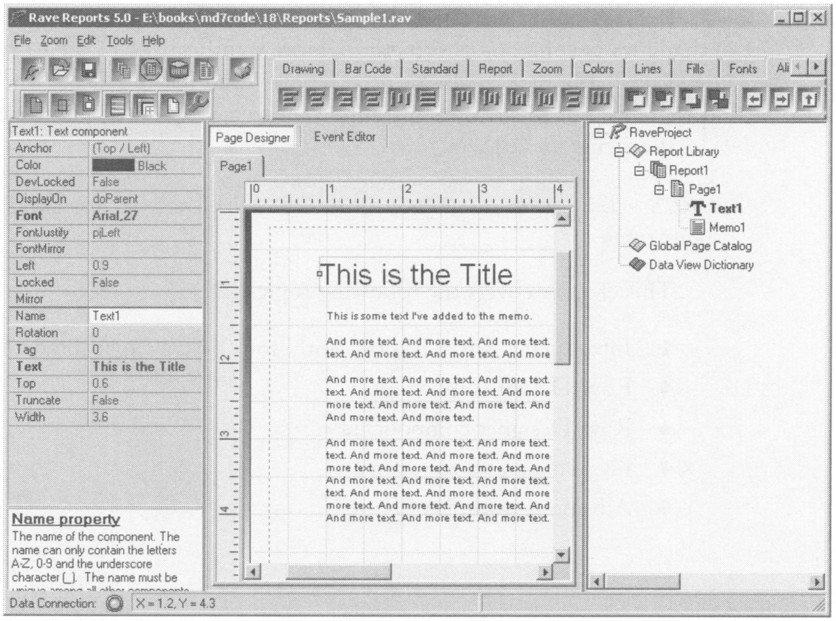
NOTE *A key feature of Rave—and one of the reasons Borland chose it over other solutions—is that it is a completely cross-platform solution you can use on both Windows and Linux. Not only do Rave components integrate with both the VCL and CLX, but the Rave Designer is itself a cross-platform application written in CLX.*

Rave: The Report Authoring Visual Environment

To start the Rave visual report design environment, double-click on a `TRvProject` component dropped on a form or select Tools ➢ Rave Designer in the Delphi IDE. When you activate this environment, you'll see a window like that shown in Figure 18.1. As you can see, the Rave Designer includes several sections: the Page Designer, Event Editor, Property panel, Project Tree panel, toolbars, Toolbar Palette, and status bar.

FIGURE 18.1

The Rave Designer with a simple report



Text1: Text component

Anchor	(Top / Left)
Color	■ Black
DevLocked	False
DisplayOn	doParent
Font	**Arial,27**
FontJustify	pjLeft
FontMirror	
Left	0.9
Locked	False
Mirror	
Name	Text1
Rotation	0
Tag	0
Text	**This is the Title**
Top	0.6
Truncate	False
Width	3.6

Name property

The name of the component. The name can only contain the letters A-Z, 0-9 and the underscore character (_). The name must be

X = 1.2, Y = 4.3

This is the Title

This is some text I've added to the memo.

And more text. And more text. And more text.
text. And more text. And more text. And more

And more text. And more text. And more text.
text. And more text. And more text. And more
more text. And more text. And more text. And
And more text. And more text.

And more text. And more text. And more text.
text. And more text. And more text. And more
more text. And more text. And more text. And
And more text. And more text. And more text.
text. And more text. And more text. And more
more text. And more text. And more text. And
And more text. And more text. And more text.

NOTE *Any time you want to see the result of your efforts, press F9 in the Rave Designer for a preview of the current report.*

Keep in mind that Rave allows your end users to create or modify their own reports. You can set the Rave Designer level to Beginner, Intermediate, or Advanced in the Edit ➢ Preferences dialog box (in the Environment section) so your end users will be working at the level they're comfortable with and won't have more power than you want to give them. You can also lock features of the report so they cannot be modified.

PAGE DESIGNER AND EVENT EDITOR

The central portion of the Rave Designer window hosts the Page Designer (where you lay out the report) and the Event Editor (where you can provide scripts to customize the report at run time).

The Page Designer is the most noticeable aspect of Rave. This page is the foundation of a report, where you perform all the designing actions. The page displays a grid pattern, although you can change the look and feel of the page with the preference settings. The names of the current pages being designed appear in the tabs above the Page Designer (*Page1* in the figure).

The Event Editor allows you to define custom scripting code for the report components. Each component has several different types of events that can be used for calculations, string manipulation, or custom report logic. The Event Editor is an advanced feature of Rave; I'll cover it briefly at the end of this chapter.

THE PROPERTY PANEL

The Property panel at left in the Rave Designer helps you customize the way components appear or behave. This panel has a role similar to that of Delphi's Object Inspector: When a component is selected on the page, the Property panel reflects the selection by displaying the different properties associated with that component. If no component is selected, the Property panel is blank.

Also as in the Delphi IDE, you can change a property value by editing the contents of the edit box, selecting an option from a drop-down list, or bringing up an editor dialog. You can double-click on any property that has a list of choices (instead of clicking on the down arrow button and selecting the option) to advance to the next item in the list.

THE PROJECT TREE PANEL

The Project Tree panel on the right side of the designer is very informative. It also provides an easy way to navigate a report project's structure. The Project Tree contains three main nodes: Report Library, Global Page Catalog, and Data View Dictionary:

Report Library Contains all the reports within the project. Each report has one or more pages. Each of those pages will normally include one or more components.

Global Page Catalog Manages reporting templates. The reporting templates can contain one or more components and can then be reused via Rave's mirroring technology. They can include items such as letter headings and footers, pre-printed forms, watermark designs, or complete page definitions that can be the foundation for other reports. You can think of the Global Page Catalog as a repository—a central location for you to store reporting items you want to be able to access from multiple reports.

Data View Dictionary Defines all the data views and other data-related objects for reports.

TOOLBARS AND THE TOOLBAR PALETTE

As in Delphi, Rave includes two types of toolbars: component toolbars and IDE toolbars. However, unlike in Delphi (which uses the tabbed Component Palette exclusively for components), in Rave you can place either type of toolbar in the open area at the top of the designer or dock them in the tabbed Toolbar Palette. At first this process can be confusing, but after you arrange the toolbars to suit your needs (using the Dock and Undock shortcut menu commands, and not by dragging them as you'd expect) you'll probably find it quite flexible.

The default component toolbars in Rave are Standard, Drawing, Report, and Bar Code. I'll describe the component toolbars in more detail later in this chapter. For the moment, suffice to say that other component toolbars may be present if you've installed add-on packages in the Rave Designer, and that components can be rearranged in the toolbars.

Editor toolbars have the ability to change or modify the project or existing components. Here is summary of the commands they host:

Project Toolbar Similar to a component toolbar, because it lets you create new report, page, and data object components in the report project. The Project toolbar can also be used to create new report projects, save and load existing report projects, and send the current report to the printer or preview.

Alignment Toolbar Has numerous tools for aligning and positioning components on a page. The first component selected often determines the benchmark for an alignment operation. You can change the printing order of components using the order buttons: Move Forward, Move Behind, Bring to Front, and Send to Back. The tap buttons let you move components in very small increments.

Fonts Toolbar Can be used to change font attributes such as Name, Size, Style, and Alignment.

Fills Toolbar Provides the fill style for shapes such as rectangles and circles.

Lines Toolbar Allows you to change the border widths and styles of lines and borders.

Colors Toolbar Determines primary and secondary colors (normally foreground and background, respectively). The left mouse button selects the primary color, and the right mouse button selects the secondary color. Eight custom color boxes are available for commonly used custom colors. Double-clicking on the primary or secondary color box at right on the toolbar opens the Color Editor dialog, which provides additional color selection tools.

Zoom Toolbar Provides many tools that allow the Page Designer to zoom in or out for easier editing.

Designer Toolbar Lets you customize the Page Designer and the Rave Designer through the preferences dialog.

THE STATUS BAR

At the bottom of the Rave Designer is the status bar. The status bar provides information about the direct data view connection status and the mouse position and sizing. The data connection LED's color tells you the status of the Rave data system (DirectDataViews): gray and green indicate an inactive or active connection, respectively; yellow and red indicate specific data access situations (respectively, waiting for response or a time-out).

The X and Y values are the coordinates of the mouse pointer in page units. When you're dropping a component on the page, if you don't release the mouse button, the size of the component will be shown by the dX and dY values (*d* stands for *delta*).

Using the RvProject Component

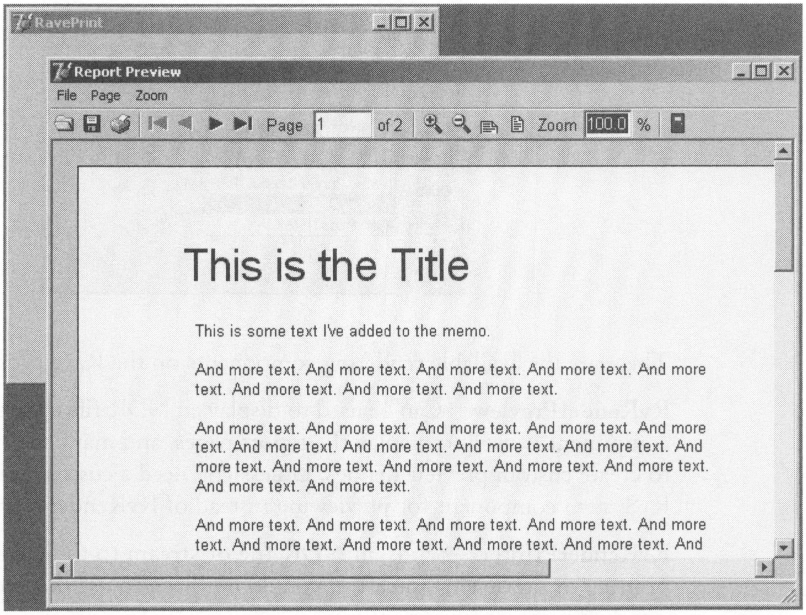
Figure 18.1 shows a trivial Rave report, which is stored in a `.rav` file. To connect this report to your Delphi 7 application you use the Rave page, which provides a series of components. The central component is RvProject. Place this component on a form or data module, set its `ProjectFile` property to a Rave file, and write this event handler code for a button:

```
RvProject1.Execute;
```

You now have a working application (the RavePrint example among the source code files) that can print a report and that includes an embedded print preview feature, as you can see in Figure 18.2. The referenced file should be distributed separately, and can be modified without having to change the Delphi program. As an alternative, you can also embed the `.rav` file into the Delphi executable by loading it in the DFM file. To accomplish this, use the `StoreRAV` property of the Rave project component.

FIGURE 18.2

The Rave Report Preview window for a report



This is the Title

This is some text I've added to the memo.

And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text.

And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text.

And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And more text. And

NOTE *As I mentioned earlier, the RvProject component, as well as all the components available in Rave Reports, can be used in either a VCL or a CLX application. The entire Rave engine is fully cross-platform.*

To control the most important settings of the report and the preview, you can hook an RvNDRWriter or an RvSystem component to the `Engine` property of the RvProject component:

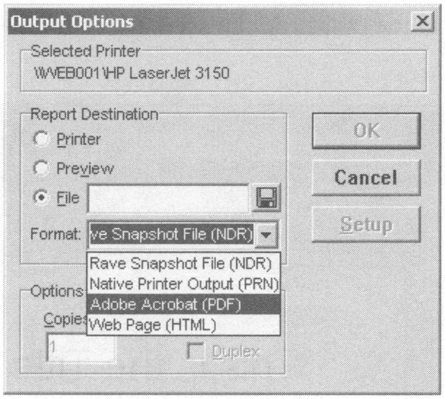
RvNDRWriter Component Generates an NDR formatted file when the report is executed. NDR formatted files are of a proprietary binary format and store all information needed by the rendering components to reproduce the report in a wide variety of formats.

RvSystem Component Combines the RvNDRWriter component with a standard print, preview, and rendering user interface. Many properties are available to customize the user interface, and override events can be defined to replace the standard dialogs with custom versions.

Rendering Formats

The Rave engine produces an NDR file or stream, generated by the RvNDRWriter. Rave's rendering engine can convert this internal representation to a wide variety of formats. To let a user choose one of the target file formats, drop the desired render components within a form of your Delphi program. When running the RvProject component's `Execute` method (as in the RavePrint example), you can choose one of the file formats in the dialog box displayed by Rave and visible in Figure 18.3.

FIGURE 18.3
After executing a Rave project, a user can choose the output format or rendering engine.



Selected Printer: \\WEB001\HP LaserJet 3150

Report Destination: ● File

Format: ve Snapshot File (NDR)

Copies: 1

These are the available rendering components on the Rave page of Delphi's Component Palette:

RvRenderPreview Can be used to display an NDR file or stream to the screen. A ScrollBox component is used to display the report pages, and many methods and properties are available to create custom preview dialogs. Unless you need a custom print preview, you should use the RvSystem component for previewing instead of RvRenderPreview.

RvRenderPrinter Sends an NDR file or stream to the printer. Unless you need a custom printing or previewing interface, you do not need to use this component—RvSystem provides standard printing functionality.

RvRenderPDF Converts an NDR file or stream to PDF (Adobe Acrobat) form. You can use properties and events to customize the type of output that is created and to provide compression support.

RvRenderHTML Converts an NDR file or stream to HTML (DHTML) form. Each page is rendered to a separate page and can be combined with templates for greater output control.

RvRenderRTF Converts an NDR file or stream to RTF form. The RTF format uses field areas for accurate positioning and reproduction of the report.

RvRenderText Converts an NDR file or stream to text form. Many graphical commands and components, such as lines and rectangles, are ignored during text rendering.

In addition to letting a user choose a file format in the dialog, rendering can be done programmatically. For example, to convert a report to a PDF file directly, you can write the following code (taken again from the RavePrint example):

```
procedure TFormRave.btnPdfClick(Sender: TObject);
begin
  RvSystem1.DefaultDest := rdFile;
  RvSystem1.DoNativeOutput := False;
  RvSystem1.RenderObject := RvRenderPDF1;
  RvSystem1.OutputFileName := 'Simple2.pdf';
  RvSystem1.SystemSetups := RvSystem1.SystemSetups - [ssAllowSetup];
  RvProject1.Engine := RvSystem1;
  RvProject1.Execute;
end;
```

Data Connections

Data connection components provide a link between data contained in a Delphi application and the DirectDataViews available in the Rave Designer. Notice that the value defined in the `Name` property for each data connection component is used to provide the link with the Rave report. For this reason, take care to avoid changing the component names after the DirectDataViews are created in Rave.

The data connection components are as follows:

RvCustomConnection Provides data to Rave reports using programmed events and can be used to send non-database data to a visual report.

RvDataSetConnection Connects any `TDataSet` class descendent component to a Rave DirectDataView. Using the `FieldAliasList` property, you can also modify the names of the dataset fields, replacing them with more legible names for the developers or end users building the report. If you need sorting or filtering for lookups or master-detail relationships within the Rave reports, you can handle the `OnSetSort` and `OnSetFilter` events.

RvTableConnection and RvQueryConnection Connect BDE Table and Query components to a Rave DirectDataView. Rave natively provides sorting and filtering support for table connections.

As a first example of how to construct a database-related report, I've created the RaveSingle example, which is an update of the DbxSingle program (built in Chapter 14, "Client/Server with dbExpress") with a Rave project and a connection:

```
object RvDataSetConnection1: TRvDataSetConnection
  RuntimeVisibility = rtDeveloper
```

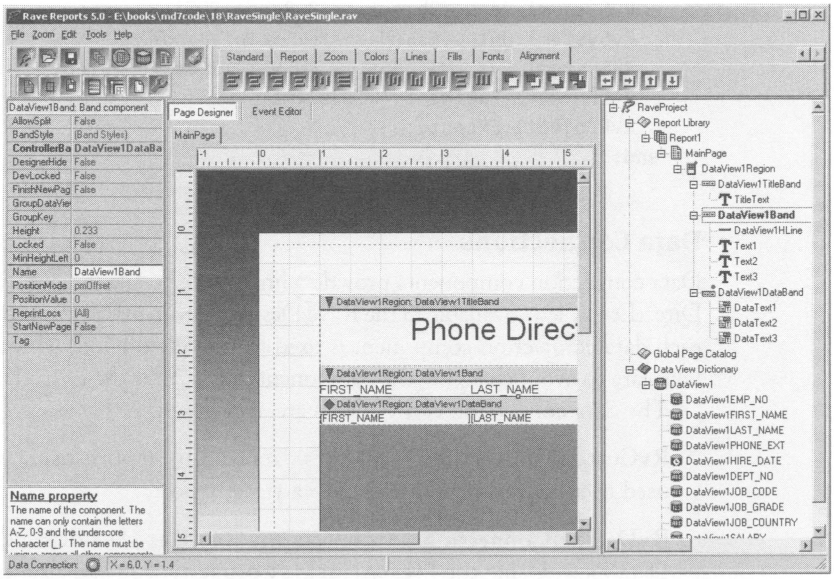
```
    DataSet = SimpleDataSet1
  end
  object RvProject1: TRvProject
    ProjectFile = 'RaveSingle.rav'
  end
```

In the Rave Designer, I created a new project, the file `RaveSingle.rav` in the `RaveSingle` folder. To refer to the data exposed by the Delphi program, you need to add a data view by clicking the New Data Object button on the Project toolbar, selecting the Direct Data View option (more details on the other alternatives later), and choosing an available connection. The list you'll see depends on the connections available in the project currently active in the Delphi IDE.

You can now create a report with the help of a wizard. From the Rave Designer menu, choose Tools ➢ Report Wizards ➢ Simple Table. Select the data view (you should have only one if you've followed these steps), and in the following window choose the dataset fields you want to include in your report. You should see a report like the one shown in Figure 18.4.

FIGURE 18.4

The RaveSingle report (generated with the help of a wizard) at design time



DataView1Band: Band component

AllowSplit	False
BandStyle	(Band Styles)
ControllerBa	**DataView1DataBa**
DesignerHide	False
DevLocked	False
FinishNewPag	False
GroupDataVie	
GroupKey	
Height	0.233
Locked	False
MinHeightLeft	0
Name	DataView1Band
PositionMode	pmOffset
PositionValue	0
ReprintLocs	(All)
StartNewPage	False
Tag	0

Name property
The name of the component. The name can only contain the letters A-Z, 0-9 and the underscore character (_). The name must be

Data Connection: X = 6.0, Y = 1.4

Rave Reports 5.0 - E:\books\md7code\18\RaveSingle\RaveSingle.rav
File Zoom Edit Tools Help
Standard Report Zoom Colors Lines Fills Fonts Alignment
Page Designer Event Editor
MainPage
DataView1Region: DataView1TitleBand
Phone Direc
DataView1Region: DataView1Band
FIRST_NAME LAST_NAME
DataView1Region: DataView1DataBand
FIRST_NAME LAST_NAME
RaveProject
Report Library
Report1
MainPage
DataView1Region
DataView1TitleBand
TitleText
DataView1Band
DataView1HLine
Text1
Text2
Text3
DataView1DataBand
DataText1
DataText2
DataText3
Global Page Catalog
Data View Dictionary
DataView1
DataView1EMP_NO
DataView1FIRST_NAME
DataView1LAST_NAME
DataView1PHONE_EXT
DataView1HIRE_DATE
DataView1DEPT_NO
DataView1JOB_CODE
DataView1JOB_GRADE
DataView1JOB_COUNTRY

As you can see the in the Project Tree, the wizard generated a report page with a Region containing three Band components: one for the title, one for the table header, and a data-aware band for the items. I'll cover the details of using these components in the section "Regions and Bands," later in this chapter; for the moment, I wanted to build and let you experiment with a working example.

Components of the Rave Designer

The previous section discussed the Rave components available in Delphi, because when you use this reporting engine you'll do most of your work in the report designer. The core element of a report (available at the top of the Project Tree view) is the RaveProject component, also called the Project Manager. This is not a component you drop on a report, because there is always only one object of this type in every report (as there is one global `Application` object in a Delphi program).

The RaveProject component is the owner of all the other report components. Like every component in Rave, the Project Manager has properties. To see them, select RaveProject in the Project Tree, and then look in the Property panel. To create a new RaveProject, click the New Project button on the Project toolbar. Doing so will create a new file, as well as the new Project Manager.

Every project can host multiple reports, represented by the Report component. A Report component contains the pages of a report. There can be multiple Reports in one project, and each Report can have multiple Pages. The Page component is the base visual component upon which you can place the report's visual components. You complete the designing and layout of a report here.

In addition to the list of reports available under the Report Library node, a Rave project has a Global Page Catalog, introduced earlier, and a Data View Dictionary. The Data View Dictionary is a detailed list of the data conduits made available by the hosting Delphi application (using the Delphi Rave connection components covered earlier) or activated directly from the report to a database.

You design the report by placing visual components directly on the page or on another container, such as a Band or a Region. Some of these components are not connected with database data—for example, the Text, Memo, and Bitmap components in the designer's Standard toolbar. Other components can be connected with a field in a database table (or are *data-aware*, to use a Delphi term), such as the DataText and DataMemo components on the Report toolbar.

Basic Components

The Standard toolbar contains seven components: Text, Memo, Section, Bitmap, Metafile, FontMaster, and PageNumInit. Many of the standard components are used frequently when designing reports.

TEXT AND MEMO COMPONENTS

The Text component is useful for displaying a single line of text on the report. It acts like a label that can contain simple text (not data). When placed on the report, the Text box is surrounded by a box that indicates its boundaries.

Memo components are similar to Text components, but they can contain multiple lines of text. When you set the Memo's font, all text in the Memo will have the same font, as in a Delphi Memo component. Once you've entered the text, the Memo box can be resized, and the text within the Memo box will be repositioned accordingly. If text seems to missing from the Memo box after it has been entered in the Memo Editor, resize the box to allow all text to be visible.

THE SECTION COMPONENT

The Section component is used to group components, like a Panel in Delphi. It provides advantages such as letting you move all components that form part of the Section with one mouse click, as opposed to making you move each component individually or try to select all components before moving.

The Project Tree is helpful when you're dealing with the Section component. From an expanded node, it is easy to see what components are in each Section, because compound components can form a tree with parent-child relationships.

TIP *The Section component is important when you're mirroring, which allows visual design inheritance within your report designs.*

GRAPHICAL COMPONENTS

The Bitmap and Metafile components let you place images in a report. Bitmap supports raster image files with the extension `.bmp`, and Metafile supports vector image files with the extensions `.wmf` and `.emf`. Metafiles are not supported when you use Rave in a CLX application, because they are based on a specific Windows technology.

THE FONTMASTER COMPONENT

Each Text component in a report has a `Font` property. By setting this property, you can assign a specific font to the component. In many cases, it may be useful and necessary to set the same font properties for more than one object. Although you can do so by selecting more than one component at a time, this method has a drawback: You have to keep track of which fonts must be the same typeface, size, and style, which is not an easy task for more than one report.

The FontMaster component lets you define standard fonts for different parts of the report, such as the headers, body, and footers. The FontMaster is a non-visual component (designated by the button's green color), so there will be no visual reference to it on the page (unlike in Delphi); and, like other non-visual components, it can only be accessed using the Project Tree.

Once you've set the FontMaster component's `Font` property, linking it to a body of text is simple. On the report, select a Text/Memo component, and then use the down arrow button by the `FontMirror` property in the Property panel to choose a FontMaster link. Any component whose `FontMirror` property is set to the FontMaster will be affected by and changed to the FontMaster's `Font` property.

When you set a component's `FontMirror` property, the component's `Font` property will be overridden by the FontMaster's `Font` property. Another side effect of using the FontMaster is that the Font toolbar is disabled when the `FontMirror` property is set for that component.

There can be more than one FontMaster per page; however, it is good practice to rename FontMaster components to describe their function. You should also locate them on a Global Page, so they can be used by all reports within the project and provide a more consistent typographical layout.

PAGE NUMBERS

PageNumInit is a non-visual component that lets you restart page numbering within a Report. Its use is similar to that of other non-visual components. You use PageNumInit when more advanced formatting is required.

For example, consider a customer statement report for a checking account. The account statements customers receive every month may vary in the number of pages. Suppose the first page defines the account summary page layout, the second defines the customer's credits/deposits, and the third defines withdrawals and debits. The first two reports may need only one page; but if account activity is high for a customer, then the withdrawals section may be several pages long. If the user producing the report wants each section's pages numbered individually, the summary and deposits pages should both be marked "1 of 1". If an active customer account has three pages of withdrawals and debits, this section of the statement should be numbered "1 of 3", "2 of 3", and "3 of 3". PageNumInit comes in handy for this kind of page numbering.

DRAWING COMPONENTS

Like the standard components, drawing components aren't related to data. Rave reports can include three components for lines: Generic lines are drawn in any direction and include angled lines; horizontal and vertical lines have a fixed direction. Available geometric shapes include squares, rectangles, circles, and ellipses. You can drop a shape on a report and then move it behind another element. For example, you can place a rectangle around a DataBand, set it so that it is sized to completely fill the band, and then move it behind the other components placed in the band.

BAR CODE COMPONENTS

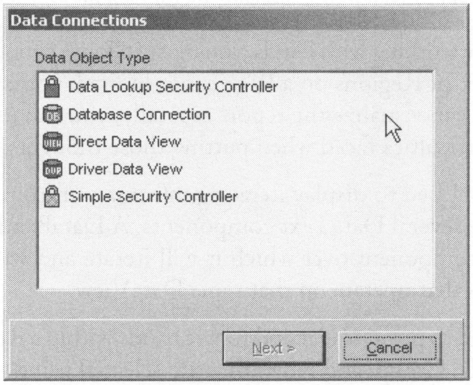
Bar code components are used to create many different kinds of bar codes in a report. Bar codes are for users who know exactly what they need, you must have background knowledge about bar codes and how they are used. To define a bar code value, go to the Property panel and type the value in the `Text` property box.

Rave bar codes support includes the following:

- POSTNET (POSTal Numeric Encoding Technique) bar code, specifically used by the U.S. Postal Service
- I2of5BarCode (interleaved 2 of 5), for numeric information only
- Code39BarCode, an alphanumeric bar code that can encode decimal numbers, the uppercase alphabet, and a few special symbols
- Code128BarCode, a high-density alphanumeric bar code designed to encode all 128 ASCII characters
- UPCBarCode (Universal Product Code), which has a fixed length of 12 digits and was designed for coding products
- EANBarCode (European Article Numbering System), which is identical to the UPC but has 13 digits, 10 numeric characters, 2 country code characters, and a check digit

Data Access Objects

In most cases, a report will be based on a data feed, either directly from a database or from a Delphi application. The data might come from a dataset connected to a database or from some internal processing of a Delphi program. When you choose the New Data Object button on the Project toolbar, you'll see the alternatives shown here and described in the following list:



RaveDatabase Component (Database Connection) Provides database connection parameters for the DriverDataView component. Only database connections for which DataLink drivers are installed will be allowed.

RaveDirectDataView Component (Direct Data View) Provides a means to retrieve data from a data connection component located within the hosting Delphi application, as in the last example. (The selection is called Direct Data View even if this is not the direct database connection from the report, but is instead the indirect connection based on data fetched from the database by a hosting application.)

RaveDriverDataView Component (Driver Data View) Provides a means to define a query for a specific database connection using a query language such as SQL. A query builder will be displayed to define the query.

RaveSimpleSecurity Component (Simple Security Controller) Implements the most basic form of security by using a simple list of username and password pairs in the `UserList` property. `UserList` contains one username and password pair per line in this format: `Username = password`. `CaseMatters` is a Boolean property that controls whether the password is case sensitive.

RaveLookupSecurity Component (Data Lookup Security) Allows username and password pairs to be checked against entries in a database table. The `DataView` property specifies the DataView to use to look up the username and password. The `UserField` and `PasswordField` properties are used to look up the username and the password to verify against.

Regions and Bands

A Region component is a container for Band components. In its simplest form, the Region could be the whole Page component. This would be true for a report that is a list type. Many master-detail reports could be made to fit a single-Region design. However, do not be limited by thinking of a Region as being the entire Page; the properties for a Region deal with its size and location on the Page. Creative use of Regions will give you more flexibility when you're designing complex reports. Multiple Regions can be placed on a single Page; they can be side-by-side, one above the other, or staggered about the Page.

TIP Do not *confuse a Region with a Section. Region components contain Bands and only Bands. A Section component can contain any group of components, including Region components, but cannot directly contain Bands.*

When you're working with Bands, you must follow a simple rule: Bands must be in a Region. Notice that the number of Regions on a Page is not limited, nor is the number of Bands within a Region. As long as you can visualize the report mentally, you can use a combination of Regions and Bands to solve any difficulties faced when putting those thoughts into design. There are two band types:

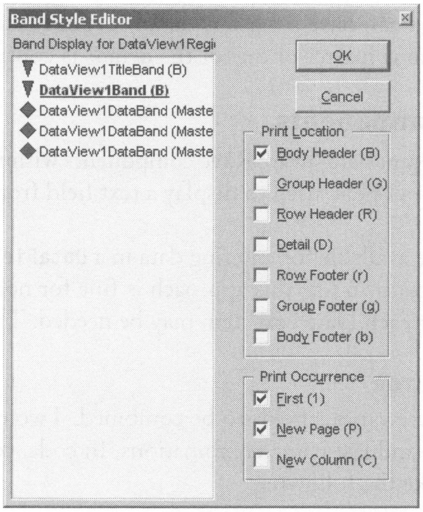
DataBand Used to display iterating information from a DataView. In general, a DataBand will contain several DataText components. A DataBand's `DataView` property must be set to the DataView component over which it will iterate and will typically contain other data-aware components that operate on that same DataView.

Band Used to print header and footer bands within a Region. Supported header and footer types include Body, Group, and Row; they are selected using the `BandStyle` property. Page headers and footers are not needed in a Band because you can drop them directly on the page outside the Region component.

`ControllerBand` is an important property for the Band component. This property determines which DataBand a Band belongs to (or is controlled by). When the controlling DataBand has been set, notice that the graphics symbol on the Band points in the direction of that controlling Band and that the color of the symbols matches. The letter codes shown on the Band are explained in the following section, "The Band Style Editor."

THE BAND STYLE EDITOR

Go to the `BandStyle` property of a Band or DataBand component and click the ellipsis button to open the Band Style Editor:


Band Style Editor

Band Display for DataView1Regi
- ▼ DataView1TitleBand (B)
- ▼ **DataView1Band (B)**
- ◆ DataView1DataBand (Maste
- ◆ DataView1DataBand (Maste
- ◆ DataView1DataBand (Maste

Print Location
- ☑ Body Header (B)
- ☐ Group Header (G)
- ☐ Row Header (R)
- ☐ Detail (D)
- ☐ Row Footer (r)
- ☐ Group Footer (g)
- ☐ Body Footer (b)

Print Occurrence
- ☑ First (1)
- ☑ New Page (P)
- ☐ New Column (C)

This editor provides a simple method to select the features you want for a Band by using the check boxes. Note that a Band can have several different features active at a time. This means it is possible for a Band to be both a Body Header and Body Footer simultaneously.

The display area in the Band Style Editor is designed to represent the flow of a report in pseudo-layout style. DataBands are duplicated three times to show that they are repeated. The current Band that is being edited is highlighted: Although you see the other Bands in the editor, you can only modify the settings of the current one.

The Band Style Editor uses symbols and letters in the display area and in the Page Layout area (for example, see Figure 18.6) to inform you about each Band's behavior. The major difference between these two representations is that the Band Style Editor display arranges the bands in a pseudo flow according to the definition of each Band. In the Band Style Editor the Bands are arranged according to the logical flow and the order they are placed in the report at design time. The Bands sequence in the report's output is mainly controlled by this order.

Headers (capital letters *BGR*, which stand for Body, Group, and Row, respectively) will print first, followed by the DataBand, and then the footers (lower case letters *bgr*) for each level. However, if more than one header is defined for a particular level, then the header Bands are processed in the order they are arranged in the Region. So, it is possible to put all the headers at the top, all the DataBands in the middle, and all the footers at the bottom of a Region for all levels of a master-detail report. Alternatively, you can group each level, with the appropriate headers, footers, and DataBands together for each level. Rave lets you use the Region layout in a way that makes the most sense to the design flow. Just remember that the order of precedence for Bands at the same level is controlled by their order within the Region.

Two symbols show the parent/child or master/detail relationships of the various bands:

- The *triangle* symbol (up/down arrows) indicates that the band is controlled by a master band with the same color (level), which can be found in the direction of the arrow.
- The *diamond* symbol represents a master or controlling band.

These symbols are both color-coded and indented to represent the level of master-detail flow. Remember that you can have a master/detail/detail relationship in which both details are both controlled by the same master or one of the details is controlled by the other detail.

Data-Aware Components

You can place different data-aware Rave components within a DataBand. The most common choice is the DataText component, used to display a text field from a dataset, as you saw in the RaveSingle example.

Two options are available for entering data in a `DataField` property. The first is to select a single field using the drop-down list; this approach is fine for normal database reporting where only a single data field for each DataText item may be needed. The second is to use the Data Text Editor.

THE DATA TEXT EDITOR

Many reports require various fields to be combined. Two common examples are city, state, and zip code, or first name and last name combinations. In code, you can accomplish these combinations using statements like the following:

```
City + ', ' + State + ' - ' + Zip
FirstName & LastName
```

The `DataField` property has a Data Text Editor, shown in Figure 18.5, which assists you in building composite fields. Click on the ellipsis to open the DataText Editor; it gives you the power to concatenate fields, parameters, or variables to build a complex data-aware text field by selecting items from the list boxes. This editor includes a lot of combinations; I will cover them quickly here, and you can try them in practice.

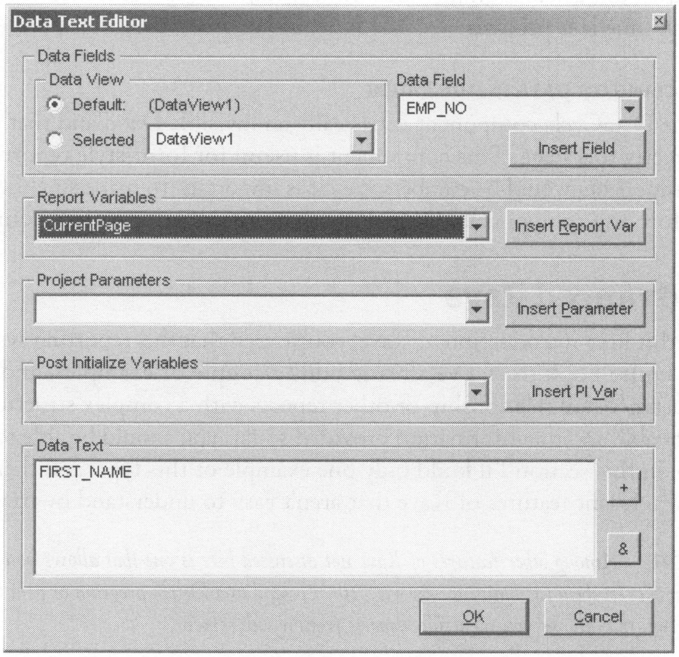
Note that the dialog box is divided into five groups: Data Fields, Report Variables, Project Parameters, Post Initialize Variables, and Data Text. Data Text is the result window: Watch this window when you're inserting items. The two buttons on the right side of this window are plus (+) and ampersand (&). The + button adds two items together with no spaces, and the & button concatenates them with a single space (as long as the previous field was not blank). So, the first step is to decide whether you want to use + or &, and then select the text from one of the three groups above the Data Text window.

The DataText component is not limited to printing database data: You can also use project parameters and report variables. Go to the Report Variables group, look in the list box, note the variables that are available. The Project Parameters list could contain `UserName`, `ReportTitle`, or

`UserOption` parameters initialized by the application. To create the list of project parameters, select the Project node in the Project Tree (the top item). Then, in the Properties panel, click the ellipsis button by the Parameters property to open typical string editor, where you can enter the different parameters you want to pass to Rave from the application (such as the `UserName`).

FIGURE 18.5

The Rave Designer's Data Text Editor



Data View: ● Default: (DataView1)
○ Selected DataView1
Data Field: EMP_NO
Report Variables: CurrentPage
Data Text: FIRST_NAME

FROM TEXT TO MEMO

The DataMemo component displays a memo field from a DataView. The main difference between the DataMemo component and the DataText component is that the DataMemo component is used to print text that may require more than one line and will thus need to be wrapped. For example, it could be used to print remarks about a customer at the bottom of each Page of an invoice.

One use for the DataMemo component is mail-merge functions. The easiest way to accomplish this is to set the `DataView` and `DataField` properties to the source of the Memo field. Then, launch the Mail Merge Editor by clicking the ellipsis button by the `MailMergeItems` property. This editor lets you set the items in the Memo that will be changed.

To use the Mail Merge Editor, click the Add button. In the Search Token window, type the item that is in the Memo and that will be replaced. Then, either type the replacement string in the Replacement window or click the Edit button to start the Data Text Editor that will help you select different DataViews and fields.

CALCULATING TOTALS

The CalcText component is a data-aware totaling component. The main difference between the DataText component and the CalcText component is that the CalcText component is specially designed to perform calculations and display the results.

The `CalcType` property determines the type of calculation being performed; values include Average, Count, Maximum, Minimum, and Sum. For example, you can use this component to print the totals of an invoice at the top of each page.

The `CountBlanks` property determines whether blank field values are included in the Average and Count calculation methods. If `RunningTotal` is True, then the calculation will not be reset to 0 each time it is printed.

CYCLING ON DATA INSIDE PAGES

The DataCycle component is basically an invisible DataBand that adds data-iterating capabilities directly to a Page. This component is useful for form-style reports where using Regions and Band components would be cumbersome. It is important to make sure the DataCycle component appears before any components on the Page that process information from the same DataView.

Advanced Rave

In this long introduction to Rave, you've seen that this reporting system is so complex I could devote an entire book to it. I've already built a couple of examples, and I could continue by showing a master/detail relationship or other reports with a complex structure. However, with the wizards available and the information provided so far, you should be able to build similar examples yourself. So, in this section I'll build only one example of this type, and then provide information about a few relevant features of Rave that aren't easy to understand by trial and error.

NOTE *Among other features of Rave not discussed here is one that allows you to distribute the report designer to end users (to let them customize the reports), either bound to a Delphi program or as a stand-alone tool. Rave also has a server version that allows you to surface reports from a web server.*

TIP *You can learn more about these features and other aspects of Rave on Nevrona's website. Specifically, you should browse the tip collection located at* `www.nevrona.com/rave/tips.shtml`.

Master/Detail Reports

To create a master/detail report in Rave, you need two datasets in the corresponding Delphi application, but these datasets don't need to have a master/detail relationship defined in the program—the report can define such a relationship. In the RaveDetails demo program, each of the two datasets is exposed through a Rave connection:

```
object dsDepartments: TSimpleDataSet
  Connection = SQLConnection1
  DataSet.CommandText = 'select * from DEPARTMENT'
end
object dsEmployee: TSimpleDataSet
  Connection = SQLConnection1
  DataSet.CommandText = 'select * from EMPLOYEE'
end
object RvConnectionDepartments: TRvDataSetConnection
  DataSet = dsDepartments
end
```

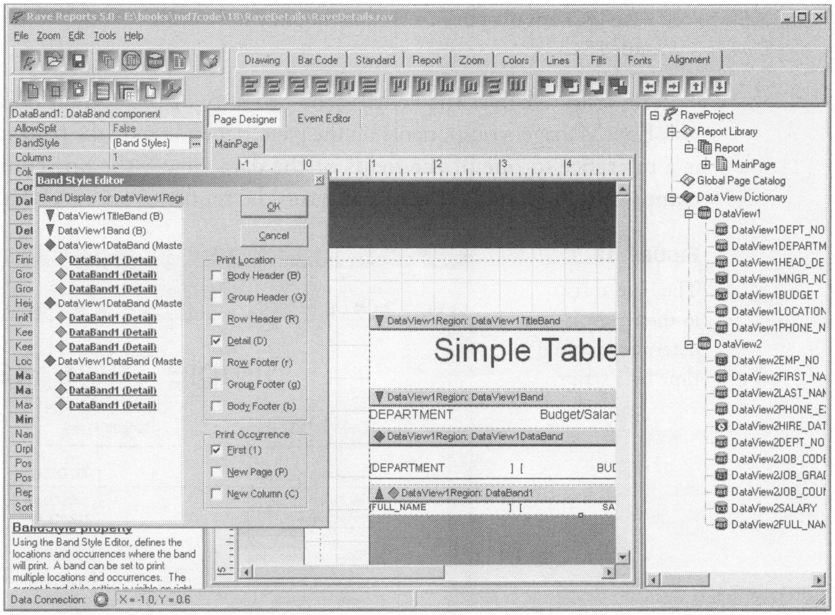
```
object RvConnectionEmployee: TRvDataSetConnection
  DataSet = dsEmployee
end
```

The report has two corresponding data views, each connected to a DataBand component (both hosted within a Region). The first DataBand, used for the main dataset, has no special settings. The secondary DataBand defines the master/detail relationship using a few properties. The `MasterDataView` property refers to the data view of the master dataset, and the `MasterKey` and `DetailKey` properties refer to the fields that define the join (in this case, they both refer to the field DEPT_NO). The `ControllerBand` property refers to the DataBand showing the data of the master dataset.

In case master/details report, the most important settings are managed by the Band Style Editor, where you must set the band as a detail. You can see this editor for the RaveDetails example in Figure 18.6. The master data view's `KeepRowTogether` property is set to True, to avoid having details end up in a different page than the master.

FIGURE 18.6

The master/detail report. The Band Style Editor appears in front of it.



DataBand1: DataBand component

AllowSplit False
BandStyle (Band Styles)
Columns 1

Band Display for DataView1Regi

- DataView1TitleBand (B)
- DataView1Band (B)
- DataView1DataBand (Maste
  - DataBand1 (Detail)
  - DataBand1 (Detail)
  - DataBand1 (Detail)
- DataView1DataBand (Maste
  - DataBand1 (Detail)
  - DataBand1 (Detail)
  - DataBand1 (Detail)
- DataView1DataBand (Maste
  - DataBand1 (Detail)
  - DataBand1 (Detail)
  - DataBand1 (Detail)

Print Location
☐ Body Header (B)
☐ Group Header (G)
☐ Row Header (R)
☑ Detail (D)
☐ Row Footer (r)
☐ Group Footer (g)
☐ Body Footer (b)

Print Occurrence
☑ First (1)
☐ New Page (P)
☐ New Column (C)

BandStyle property

Using the Band Style Editor, defines the locations and occurrences where the band will print. A band can be set to print multiple locations and occurrences. The

Data Connection: X = -1.0, Y = 0.6

File Zoom Edit Tools Help
Drawing
Bar Code
Standard
Report
Zoom
Colors
Lines
Fills
Fonts
Alignment
Page Designer
Event Editor
MainPage
Band Style Editor
OK
Cancel
DataView1Region: DataView1TitleBand
Simple Table
DataView1Region: DataView1Band
DEPARTMENT
Budget/Salar
DataView1Region: DataView1DataBand
DEPARTMENT] [BUD
DataView1Region: DataBand1
FULL_NAME] [SA
RaveProject
Report Library
Report
MainPage
Global Page Catalog
Data View Dictionary
DataView1
DataView1DEPT_NO
DataView1DEPARTM
DataView1HEAD_DE
DataView1MNGR_NC
DataView1BUDGET
DataView1LOCATION
DataView1PHONE_N
DataView2
DataView2EMP_NO
DataView2FIRST_NA
DataView2LAST_NAM
DataView2PHONE_E
DataView2HIRE_DAT
DataView2DEPT_NO
DataView2JOB_CODE
DataView2JOB_GRAD
DataView2JOB_COUN
DataView2SALARY
DataView2FULL_NAM

WARNING *To create a master/detail report, you may want to use the corresponding wizard available in Rave. In the version shipping with Delphi 7, this wizard doesn't work. At the time of this writing an update to fix this and other issues is still not available.*

Scripting Reports

At the beginning of this chapter I mentioned the Event Editor window of the Rave Designer, but so far I haven't used it. This tool is used to write code (scripts) within a report, responding to events of the various components as you'd do in Delphi. Writing scripts in the Rave Designer allows you to customize, or fine-tune, the output of a report in sophisticated ways. The language Rave uses for the scripts is Pascal based and a variation of the Delphi language, so you should be able to pick it up easily.

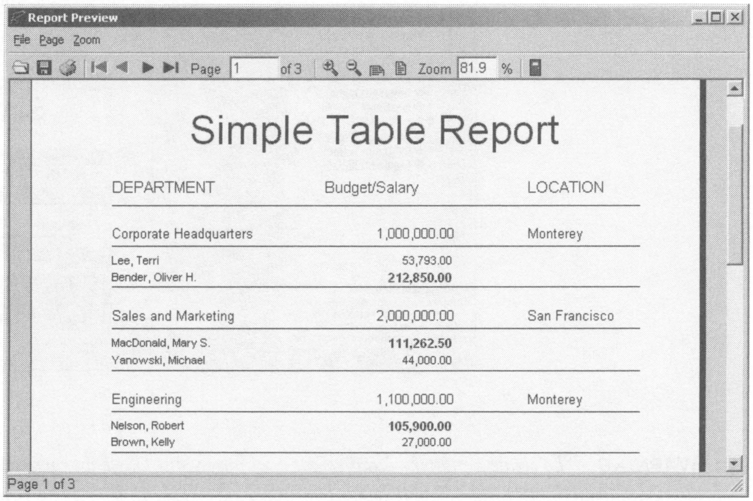
The RaveDetails example shows in bold font salaries that are greater than a specified amount. The obvious way to accomplish this is to write scripting code to be executed for each instance of the detail band—that is, for each record in the employee database. Instead of modifying the `Font` property directly, I decided to add two different FontManager components to the report page and name them to make their role understandable: `fmPlainFont` and `fmBoldFont`. You can open the report to see their properties and layout.

In the report, to highlight values outside a given range, you handle the DataText component's `BeforePrint` event. To do so, move to the Event Editor page, choose the DataText component connected to the Salary field, and select the event. In the event's code editor window, write the following code:

```
if DataView2Salary.AsFloat > 100000 then
  self.FontMirror := fmBoldFont;
else
  self.FontMirror := fmPlainFont;
end if;
```

The script changes the `FontMirror` property of the current object (`self`) to refer to one of the two FontManager components on the page, depending on the field value. Notice that `DataView2Salary` is a reference to one of the fields of the data view—the one connected to the current DataText component. Compile the script and run the report to see its effect, shown in Figure 18.7.

FIGURE 18.7
The bold text in the report is determined at run time by a script.



Simple Table Report

DEPARTMENT	Budget/Salary	LOCATION
Corporate Headquarters	1,000,000.00	Monterey
Lee, Terri	53,793.00	
Bender, Oliver H.	**212,850.00**	
Sales and Marketing	2,000,000.00	San Francisco
MacDonald, Mary S.	**111,262.50**	
Yanowski, Michael	44,000.00	
Engineering	1,100,000.00	Monterey
Nelson, Robert	**105,900.00**	
Brown, Kelly	27,000.00	

WARNING *Every time you edit a script, remember to click the Compile button; otherwise your changes won't take effect!*

Mirroring

Reporting templates can contain one or more components and can be reused via Rave's mirroring technology. The DataMirrorSection component mirrors other Sections based on the contents of a

DataField. The use of mirroring sections allows the DataMirrorSection to be very flexible. Remember that Sections can contain any other component including graphics, regions, text, and so on.

For example, you could use a DataMirrorSection component to make a single report produce different envelope formats for international or U.S. addresses. The template for international customers could include a line for the country have the text centered on the envelope, whereas the U.S. format would not include the country line would offset the text to the right of center and lower on the envelope.

Normally, one of the settings will be defined as the default. If a default is *not* defined and the field value does not match any of the other settings, then the format used will be the normal contents of the DataMirrorSection component.

Calculations to the Max

In addition to the simple CalcText component discussed earlier, the Rave Designer includes three components for handling more complex situations: CalcTotal, CalcController, and CalcOp.

CALCTOTAL

The CalcTotal is a non-visual version of the CalcText component. When this component is printed, its value is typically stored in a project parameter (defined by the `DestParam` property) and formatted according to the `DisplayFormat` property. It can be useful when you're performing totaling calculations that will be used by other calculations before being printed. If the value of the CalcTotal will only be used by other calculation components, such as CalcOp, you should leave the `DestParam` property blank.

CALCCONTROLLER

CalcController is a non-visual component that acts as a controller for CalcText and CalcTotal components through their `Controller` properties. When the controller component is printed, it signals all calculation components that it controls to perform their operation. This process allows a report to recalculate totals on group bands, detail bands, or whole pages depending upon the location of the CalcController component.

The CalcController component can also initialize a CalcText or CalcTotal component to a specific value (through the `InitCalcVar`, `InitDataField`, and `InitValue` properties). The CalcController component will initialize values only if it is used in the `Initializer` property of CalcText or CalcTotal components.

CALCOP

CalcOp is a non-visual component that allows an operation (defined by the `Operator` property) to be performed on values from different data sources. The result can then be saved in a project parameter like CalcTotal, as indicated by the `DestParam` and `DisplayFormat` properties.

For example, suppose you need to add two DataText components, as in $A + B = C$ (where A and B represent the two DataText component values and C represents the result stored in a project parameter). The three source types have many different values associated with them.

The calculation can start with different types of data sources:

- A DataField source is a field in a table, or a DataView in Rave terms. Thus in order to choose a field, you must first select a DataView.

- For a Value source, you fill in the property with a numeric value.
- A CalcVar source represents another calculation variable; you can choose one from the drop-down menu that lists all the calculation variables available in the page. This value can be from another CalcOp component or from some other calculation component.

After choosing the data sources, you select the operation to be used between them. The `Operator` property has a drop-down menu you can use to make the appropriate choice. In the example A + B = C, the operator is coAdd.

Sometimes a function needs to be performed on a value before it is processed with the second value. In this case the source's `Function` property is handy. With a function, you can converted a value (such as hours to minutes), compute a trigonometric function (like the sin of the value), or perform many others calculations (such as a square root or absolute value).

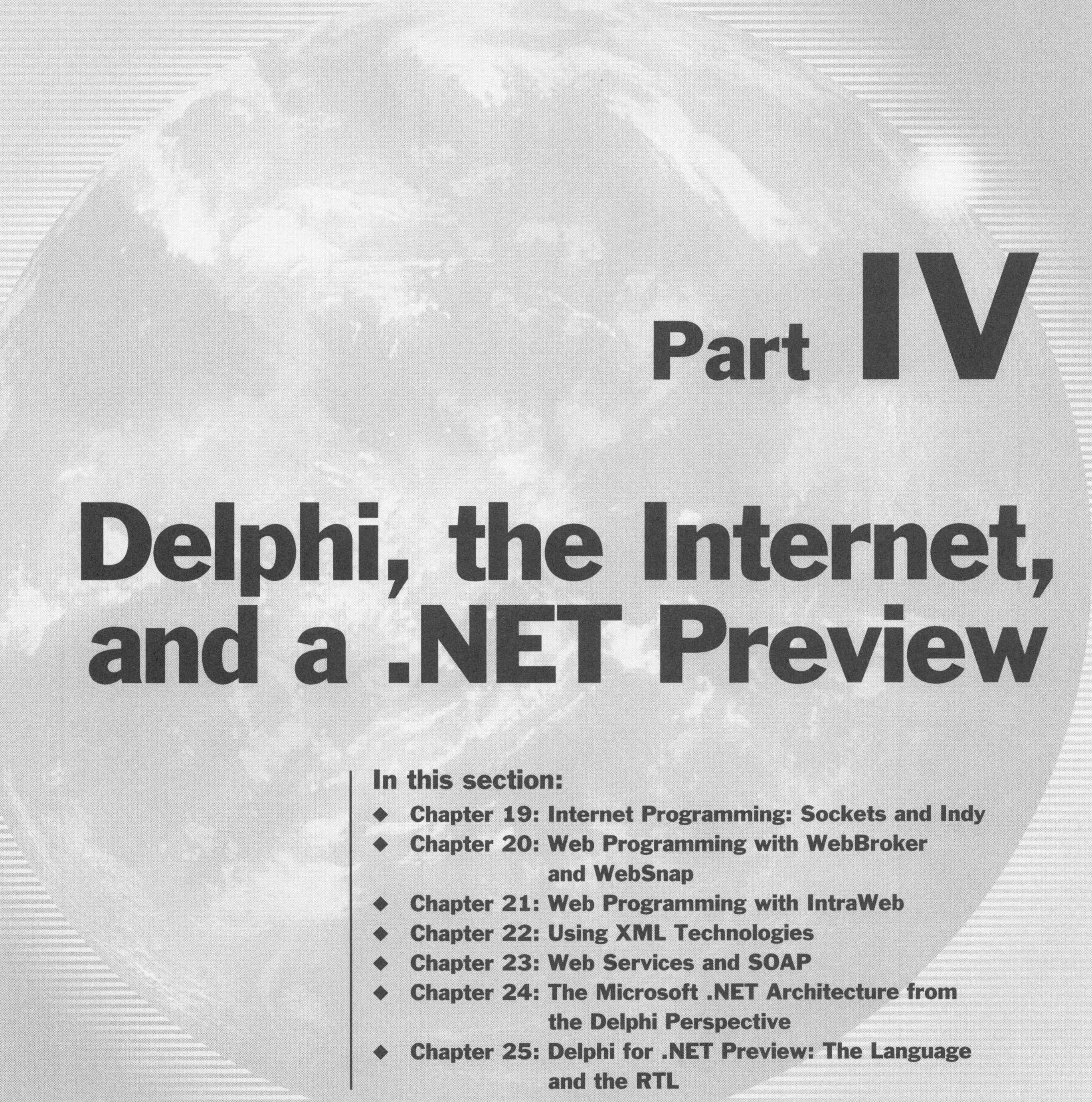
Just as it is important to do the calculations in order, it is important to make sure the components are in order in the Project Tree. A report executes components down the Project Tree. For CalcOp components or any calculation component, this means they must be in the correct order. It is also important to note that if a source value is dependent on any other components (like other CalcOp components or DataText components), those components must come first in the Project Tree.

What's Next?

This chapter has focused using Rave to add reporting capabilities to Delphi applications. I've covered basic reports and data-driven reports, describing both the components you can use in a Delphi application to connect with a report and the components you can use in the report to display data passed by the application. I also touched on some of the more advanced features of Rave, like the use of the Band Style Editor, scripting, mirroring, calculations, and a few others.

If you are looking for more documentation, refer to the PDF files that are available with Delphi but not installed with the product. These manuals and many others are available on the Delphi 7 Companion CD.

This chapter ends the book's coverage of Delphi database technologies, although you'll also use databases in the following part of the book (devoted to Internet and web programming). I'll begin by covering core technologies, such as the use of sockets, and then proceed to HTML generation, traditional Delphi web development, and the use of IntraWeb. I'll then cover the XML and SOAP technologies.



Part IV

Delphi, the Internet, and a .NET Preview

In this section:

- Chapter 19: Internet Programming: Sockets and Indy
- Chapter 20: Web Programming with WebBroker and WebSnap
- Chapter 21: Web Programming with IntraWeb
- Chapter 22: Using XML Technologies
- Chapter 23: Web Services and SOAP
- Chapter 24: The Microsoft .NET Architecture from the Delphi Perspective
- Chapter 25: Delphi for .NET Preview: The Language and the RTL

Chapter 19

Internet Programming: Sockets and Indy

With the advent of the Internet era, writing programs based on Internet protocols has become commonplace, so I've devoted five chapters to this topic. This chapter focuses on low-level socket programming and Internet protocols, Chapter 20 is devoted to server-side web programming, Chapter 21 covers IntraWeb, and Chapters 22 and 23 discuss XML and web services.

In this chapter I'll begin by looking at the sockets technology in general; then I'll move to the use of the Internet Direct (Indy) components supporting both low-level socket programming and the most common Internet protocols. I will introduce some elements of the HTTP protocol, leading up to building HTML files from database data.

Although you probably just want to use a high-level protocol, the discussion of Internet programming starts from the core concepts and low-level applications. Understanding TCP/IP and sockets will help you grasp most of the other concepts more easily.

This chapter covers the following:

- Using sockets
- Internet Direct (Indy) components
- Low-level sockets programming
- The WinInet API
- Mail protocols (SMTP and POP3)
- The HTTP protocol
- Generating HTML
- From database to HTML

Building Socket Applications

Delphi 7 ships with two sets of TCP components—Indy socket components (IdTCPClient and IdTCPServer) and native Borland components—which are also available in Kylix and are hosted in the Internet page of the Component palette. The Borland components, TcpClient and TcpServer, were probably developed to replace the ClientSocket and ServerSocket components available in past versions of Delphi. However, now that the ClientSocket and ServerSocket components have been declared obsolete (although they are still available), Borland suggests using the corresponding Indy components instead.

In this chapter I'll focus on using Indy during my discussion of low-level socket programming, not only when I cover support for high-level Internet protocols. To learn more about the Indy project, refer to the sidebar "Internet Direct (Indy) Open Source Components"; keep reading to see how you can use these components for low-level socket programming.

Before I present an example of a low-level socket-based communication, let's take a tour of the core concepts of TCP/IP so you understand the foundations of the most widespread networking technology.

INTERNET DIRECT (INDY) OPEN SOURCE COMPONENTS

Delphi ships with a collection of open-source Internet components called Internet Direct (Indy). The Indy components, previously called WinShoes (a pun on the term WinSock, the name of the Windows socket library), are built by a group of developers led by Chad Hower and are also available in Kylix. You can find more information and the most recent versions of the components at `www.nevrona.com/indy`.

Delphi 7 ships with Indy 9, but you should check the website for updated versions. The components are free and are complemented by many examples and a reasonable help file. Indy 9 includes many more components than the previous version (Indy 8, available in Delphi 6), and it has two new pages on the component palette (Indy Intercepts and Indy I/O Handlers).

With more than 100 components installed on Delphi's palette, Indy has an enormous number of features, ranging from the development of client and server TCP/IP applications for various protocols to encoding and security. You can recognize Indy components from the Id prefix. Rather than list the various components here, I'll touch on a few of them throughout this chapter.

BLOCKING AND NON-BLOCKING CONNECTIONS

When you're working with sockets in Windows, reading data from a socket or writing to it can happen asynchronously, so that it does not block the execution of other code in your network application. This is called a *non-blocking connection*. The Windows socket support sends a message when data is available. An alternative approach is the use of *blocking connections*, where your application waits for the reading or writing to be completed before executing the next line of code. When you're using a blocking connection, you must use a thread on the server, and you'll generally also use a thread on the client.

The Indy components use blocking connections exclusively. So, any client socket operation that might be lengthy should be performed within a thread or by using Indy's IdAntiFreeze component as a simpler but limited alternative. Using blocking connections to implement a protocol has the advantage of simplifying the program logic, because you don't have to use the state-machine approach of non-blocking connections.

All the Indy servers use a multithreaded architecture that you can control with the IdThreadMgrDefault and IdThreadMgrPool components. The first is used by default; the second supports thread pooling and should account for faster connections.

Foundations of Socket Programming

To understand the behavior of the socket components, you need to be confident with several terms related to the Internet in general and with sockets in particular. The heart of the Internet is the Transmission Control Protocol/Internet Protocol (TCP/IP), a combination of two separate protocols that work together to provide connections over the Internet (and that can also provide connection over a private intranet). In brief, IP is responsible for defining and routing the *datagrams* (Internet transmission units) and specifying the addressing scheme. TCP is responsible for higher-level transport services.

CONFIGURING A LOCAL NETWORK: IP ADDRESSES

If you have a local network available, you'll be able to test the following programs on it; otherwise, you can use the same computer as both client and server. In this case, as I've done in the examples, use the address 127.0.0.1 (or *localhost*), which is invariably the address of the current computer. If your network is complex, ask your network administrator to set up proper IP addresses for you. If you want to set up a simple network with a couple of spare computers, you can set up the IP address yourself; it's a 32-bit number usually represented with each of its four components (called *octets*) separated by dots. These numbers have a complex logic underlying them, and the first octet indicates the class of the address.

Specific IP addresses are reserved for unregistered internal networks. Internet routers ignore these address ranges, so you can freely do your tests without interfering with an actual network. For example, the "free" IP address range 192.168.0.0 through 192.168.0.255 can be used for experiments on a network of fewer than 255 machines.

LOCAL DOMAIN NAMES

How does the IP address map to a name? On the Internet, the client program looks up the values on a domain name server. But it is also possible to have a local *hosts* file, which is a text file you can easily edit to provide local mappings. Look at the `HOSTS.SAM` file (installed in a subdirectory of the `Windows` directory, depending on the version of Windows you have) to see a sample; you can eventually rename the file `HOSTS`, without the extension, to activate local host mapping.

You may wonder whether to use an IP or a hostname in your programs. Hostnames are easier to remember and won't require a change if the IP address changes (for whatever reason). On the other hand, IP addresses don't require any resolution, whereas hostnames must be resolved (a time-consuming operation if the lookup takes place on the web).

TCP PORTS

Each TCP connection takes place though a *port*, which is represented by a 16-bit number. The IP address and the TCP port together specify an Internet connection, or a *socket*. Different processes running on the same machine cannot use the same socket (the same port).

Some TCP ports have a standard usage for specific high-level protocols and services. In other words, you should use those port numbers when implementing those services and stay away from them in any other case. Here is a short list:

Protocol	Port
HTTP (Hypertext Transfer Protocol)	80
FTP (File Transfer Protocol)	21

Protocol	Port
SMTP (Simple Mail Transfer Protocol)	25
POP3 (Post Office Protocol, version 3)	110
Telnet	23

The `Services` file (another text file similar to the `Hosts` file) lists the standard ports used by services. You can add your own entry to the list, giving your service a name of your own choosing. Client sockets always specify the port number or the service name of the server socket to which they want to connect.

HIGH-LEVEL PROTOCOLS

I've used the term *protocol* many times now. A protocol is a set of rules the client and server agree on to determine the communication flow. The low-level Internet protocols, such as TCP/IP, are usually implemented by an operating system. But the term *protocol* is also used for high-level Internet standard protocols (such as HTTP, FTP, or SMTP). These protocols are defined in standard documents available on the Internet Engineering Task Force website (`www.ietf.org`).

If you want to implement a custom communication, you can define your own (possibly simple) protocol, a set of rules determining which request the client can send to the server and how the server can respond to the various possible requests. You'll see an example of a custom protocol later. Transfer protocols are at a higher level than transmission protocols, because they abstract from the transport mechanism provided by TCP/IP. This makes the protocols independent not only from the operating system and the hardware but also from the physical network.

SOCKET CONNECTIONS

To begin communication through a socket, the server program starts running first; but it simply waits for a request from a client. The client program requests a connection indicating the server it wishes to connect to. When the client sends the request, the server can accept the connection, starting a specific server-side socket, which connects to the client-side socket.

To support this model, there are three types of socket connections:

- *Client connections* are initiated by the client and connect a local client socket with a remote server socket. Client sockets must describe the server they want to connect to, by providing its hostname (or its IP address) and its port.
- *Listening connections* are passive server sockets waiting for a client. Once a client makes a new request, the server spawns a new socket devoted to that specific connection and then gets back to listening. Listening server sockets must indicate the port that represents the service they provide. (The client will connect through that port.)
- *Server connections* are activated by servers; they accept a request from a client.

These different types of connections are important only for establishing the link from the client to the server. Once the link is established, both sides are free to make requests and to send data to the other side.

Using Indy's TCP Components

To let two programs communicate over a socket (either on a local area network or over the Internet), you can use the IdTCPClient and IdTCPServer components. Place one of them on a program form and the other on another form in a different program; then, make them use the same port, and let the client program refer to the host of the server program, and you'll be able to open a connection between the two applications. For example, in the IndySock1 project group, I've used the two components with these settings:

```
// server program
object IdTCPServer1: TIdTCPServer
  DefaultPort = 1050
end
// client program
object IdTCPClient1: TIdTCPClient
  Host = 'localhost'
  Port = 1050
end
```

NOTE *The Indy server sockets allow binding to multiple IP addresses and/or ports, using the* `Bindings` *collection.*

As this point, in the client program you can connect to the server by executing

```
IdTCPClient1.Connect;
```

The server program has a list box used to log information. When a client connects or disconnects, the program lists the IP of that client along with the operation, as in the following `OnConnect` event handler:

```
procedure TFormServer.IdTCPServer1Connect(AThread: TIdPeerThread);
begin
  lbLog.Items.Add ('Connected from: ' +
    AThread.Connection.Socket.Binding.PeerIP);
end;
```

Now that you have set up a connection, you need to make the two programs communicate. Both the client and server sockets have read and write methods you can use to send data, but writing a multithreaded server that can receive many different commands (usually based on strings) and operate differently on each of them is far from trivial.

However, Indy simplifies the development of a server by means of its command architecture. In a server, you can define a number of commands, which are stored in the `CommandHandlers` collection of the IdTCPServer. In the IndySock1 example the server has three handlers, all implemented differently to show you some of the possible alternatives.

The first server command, called *test*, is the simplest one, because it is fully defined in its properties. I've set the command string, a numeric code, and a string result in the `ReplyNormal` property of the command handler:

```
object IdTCPServer1: TIdTCPServer
  CommandHandlers = <
```

```
    item
      Command = 'test'
      Name = 'TIdCommandHandler0'
      ParseParams = False
      ReplyNormal.NumericCode = 100
      ReplyNormal.Text.Strings = (
        'Hello from your Indy Server')
      ReplyNormal.TextCode = '100'
    end
```

The client code used to execute the command and show its response is as follows:

```
procedure TFormClient.btnTestClick(Sender: TObject);
begin
  IdTCPClient1.SendCmd ('test');
  ShowMessage (IdTCPClient1.LastCmdResult.TextCode + ' : ' +
    IdTCPClient1.LastCmdResult.Text.Text);
end;
```

For more complex cases, you should execute code on the server and read and write directly over the socket connection. This approach is shown in the second command of the trivial protocol I've come up with for this example. The server's second command is called **execute**; and it has no special property set (only the command name), but has the following **OnCommand** event handler:

```
procedure TFormServer.IdTCPServer1TIdCommandHandler1Command(
  ASender: TIdCommand);
begin
  ASender.Thread.Connection.Writeln ('This is a dynamic response');
end;
```

The corresponding client code writes the command name to the socket connection and then reads a single-line response, using different methods than the first one:

```
procedure TFormClient.btnExecuteClick(Sender: TObject);
begin
  IdTCPClient1.WriteLn('execute');
  ShowMessage (IdTCPClient1.ReadLn);
end;
```

The effect is similar to the previous example, but because it uses a lower-level approach, it should be easier to customize it to your needs. One such extension is provided by the third and last command in the example, which allows the client program to request a bitmap file from the server (in a sort of file-sharing architecture). The server command has parameters (the filename) and is defined as follows:

```
object IdTCPServer1: TIdTCPServer
  CommandHandlers = <
    item
      CmdDelimiter = ' '
      Command = 'getfile'
      Name = 'TIdCommandHandler2'
      OnCommand = IdTCPServer1TIdCommandHandler2Command
    ParamDelimiter = ' '
    ReplyExceptionCode = 0
    ReplyNormal.NumericCode = 0
    Tag = 0
  end>
```

The code uses the first parameter as filename and returns it in a stream. In case of error, it raises an exception, which will be intercepted by the server component, which in turn will terminate the connection (not a very realistic solution, but a safe approach and a simple one to implement):

```
procedure TFormServer.IdTCPServer1TIdCommandHandler2Command(
  ASender: TIdCommand);
var
  filename: string;
  fstream: TFileStream;
begin
  if Assigned (ASender.Params) then
    filename := HttpDecode (ASender.Params [0]);
  if not FileExists (filename) then
  begin
    ASender.Response.Text := 'File not found';
    lbLog.Items.Add ('File not found: ' + filename);
    raise EIdTCPServerError.Create ('File not found: ' + filename);
  end
  else
  begin
    fstream := TFileStream.Create (filename, fmOpenRead);
    try
      ASender.Thread.Connection.WriteStream(fstream, True, True);
      lbLog.Items.Add ('File returned: ' + filename +
        ' (' + IntToStr (fStream.Size) + ')');
    finally
      fstream.Free;
    end;
  end;
end;
```

The call to the `HttpDecode` utility function on the parameter is required to encode a pathname that includes spaces as a single parameter, at the reverse the client program calls `HttpEncode`. As you can see, the server also logs the files returned and their sizes, or an error message. The client program reads the stream and copies it into an Image component, to show it directly (see Figure 19.1):

```
procedure TFormClient.btnGetFileClick(Sender: TObject);
var
  stream: TStream;
begin
  IdTCPClient1.WriteLn('getfile ' + HttpEncode (edFileName.Text));
  stream := TMemoryStream.Create;
  try
    IdTCPClient1.ReadStream(stream);
```

```
      stream.Position := 0;
      Image1.Picture.Bitmap.LoadFromStream (stream);
    finally
      stream.Free;
    end;
  end;
```

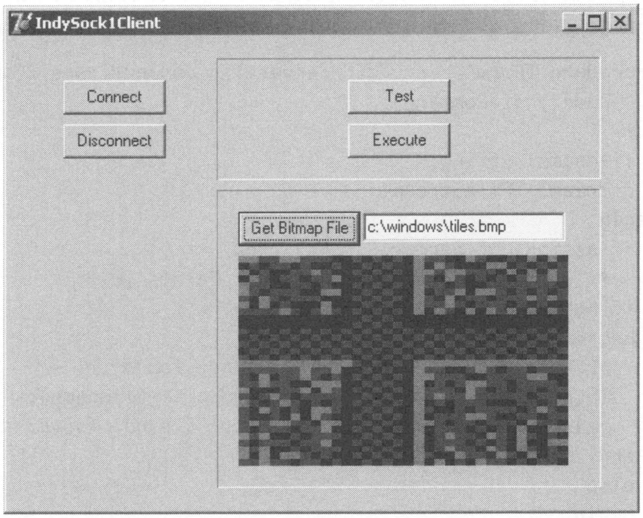


FIGURE 19.1 The client program of the IndySock1 example

Sending Database Data over a Socket Connection

Using the techniques you've seen so far, you can write an application that moves database records over a socket. The idea is to write a front end for data input and a back end for data storage. The client application will have a simple data-entry form and use a database table with string fields for Company, Address, State, Country, Email, and Contact, and a floating-point field for the company ID (called CompID).

NOTE *Moving database records over a socket is exactly what you can do with DataSnap and a socket connection component (as covered in Chapter 16, "Multitier DataSnap Applications") or with SOAP support (discussed in Chapter 23, "Web Services and SOAP").*

The client program I've come up with works on a ClientDataSet with this structure saved in the current directory. (You can see the related code in the `OnCreate` event handler.) The core method on the client side is the handler of the Send All button's `OnClick` event, which sends all the new records to the server. A new record is determined by looking to see whether the record has a valid value for the CompID field. This field is not set up by the user but is determined by the server application when the data is sent.

For all the new records, the client program packages the field information in a string list, using the structure *FieldName=FieldValue*. The string corresponding to the entire list, which is a record, is then sent to the server. At this point, the program waits for the server to send back the company ID, which is then saved in the current record. All this code takes place within a thread, to avoid blocking the user interface during the lengthy operation. By clicking the Send button, a user starts a new thread:

```
procedure TForm1.btnSendClick(Sender: TObject);
var
  SendThread: TSendThread;
begin
  SendThread := TSendThread.Create(cds);
  SendThread.OnLog := OnLog;
  SendThread.ServerAddress := EditServer.Text;
  SendThread.Resume;
end;
```

The thread has a few parameters: the dataset passed in the constructor, the address of the server saved in the `ServerAddress` property, and a logging event to write to the main form (within a safe `Synchronize` call). The thread code creates and opens a connection and keeps sending records until it's finished:

```
procedure TSendThread.Execute;
var
  I: Integer;
  Data: TStringList;
  Buf: String;
begin
  try
    Data := TStringList.Create;
    fIdTcpClient := TIdTcpClient.Create (nil);
    try
      fIdTcpClient.Host := ServerAddress;
      fIdTcpClient.Port := 1051;
      fIdTcpClient.Connect;
      fDataSet.First;
      while not fDataSet.Eof do
      begin
        // if the record is still not logged
        if fDataSet.FieldByName('CompID').IsNull or
          (fDataSet.FieldByName('CompID').AsInteger = 0) then
        begin
          FLogMsg := 'Sending ' + fDataSet.FieldByName('Company').AsString;
          Synchronize(DoLog);
          Data.Clear;
          // create strings with structure "FieldName=Value"
          for I := 0 to fDataSet.FieldCount - 1 do
            Data.Values [fDataSet.Fields[I].FieldName] :=
              fDataSet.Fields [I].AsString;
          // send the record
          fIdTcpClient.Writeln ('senddata');
          fIdTcpClient.WriteStrings (Data, True);
```

```
            // wait for reponse
            Buf := fIdTcpClient.ReadLn;
            fDataSet.Edit;
            fDataSet.FieldByName('CompID').AsString := Buf;
            fDataSet.Post;
            FLogMsg := fDataSet.FieldByName('Company').AsString +
              ' logged as ' + fDataSet.FieldByName('CompID').AsString;
            Synchronize(DoLog);
          end;
          fDataSet.Next;
        end;
      finally
        fIdTcpClient.Disconnect;
        fIdTcpClient.Free;
        Data.Free;
      end;
    except
      // trap exceptions in case of dataset errors
      // (concurrent editing and so on)
    end;
  end;
```

Now let's look at the server. This program has a database table, again stored in the local directory, with two more fields than the client application's table: LoggedBy, a string field; and LoggedOn, a data field. The values of the two extra fields are determined automatically by the server as it receives data, along with the value of the CompID field. All these operations are done in the handler of the `senddata` command:

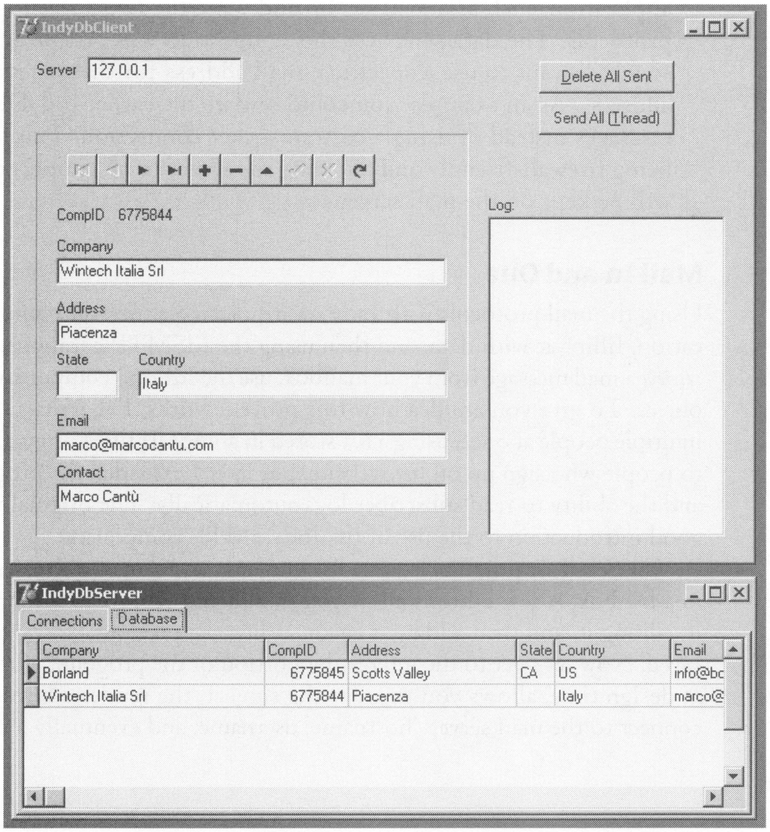
```
procedure TForm1.IdTCPServer1TIdCommandHandler0Command(
  ASender: TIdCommand);
var
  Data: TStrings;
  I: Integer;
begin
  Data := TStringList.Create;
  try
    ASender.Thread.Connection.ReadStrings(Data);
    cds.Insert;
    // set the fields using the strings
    for I := 0 to cds.FieldCount - 1 do
      cds.Fields [I].AsString :=
        Data.Values [cds.Fields[I].FieldName];
    // complete with ID, sender, and date
    Inc(ID);
    cdsCompID.AsInteger := ID;
    cdsLoggedBy.AsString := ASender.Thread.Connection.Socket.Binding.PeerIP;
    cdsLoggedOn.AsDateTime := Date;
    cds.Post;
    // return the ID
    ASender.Thread.Connection.WriteLn(cdsCompID.AsString);
  finally
    Data.Free;
  end;
end;
```

Except for the fact that some data might be lost, there is no problem when fields have a different order and if they do not match, because the data is stored in the *FieldName=FieldValue* structure. After receiving all the data and posting it to the local table, the server sends back the company ID to the client. When receiving feedback, the client program saves the company ID, which marks the record as sent. If the user modifies the record, there is no way to send an update to the server. To accomplish this, you might add a modified field to the client database table and make the server check to see if it is receiving a new field or a modified field. With a modified field, the server should not add a new record but update the existing one.

As shown in Figure 19.2, the server program has two pages: one with a log and the other with a DBGrid showing the current data in the server database table. The client program is a form-based data entry, with extra buttons to send the data and delete records already sent (and for which an ID was received back).

FIGURE 19.2

The client and server programs of the database socket example (IndyDbSock)



Server 127.0.0.1

CompID 6775844

Company
Wintech Italia Srl

Address
Piacenza

State

Country
Italy

Email
marco@marcocantu.com

Contact
Marco Cantù

Company	CompID	Address	State	Country	Email
Borland	6775845	Scotts Valley	CA	US	info@bc
Wintech Italia Srl	6775844	Piacenza		Italy	marco@

Sending and Receiving Mail

Probably the most common operation you do on the Internet is to send and receive e-mail. There is generally little need to write a complete application to handle e-mail, because some of the existing programs are rather complete. For this reason, I have no intention of writing a general-purpose mail program here. You can find some examples of those among Indy demos. Other than creating a general-purpose mail application, you can do many things with the mail components and protocols; I've grouped the possibilities into two areas:

Automatic Generation of Mail Messages An application you've written can have an About box for sending a registration message back to your marketing department or a specific menu item for sending a request to your tech support. You might even decide to enable a tech-support connection whenever an exception occurs. Another related task could automate the dispatching of a message to a list of people or generate an automatic message from your website (an example I'll show you toward the end of this chapter).

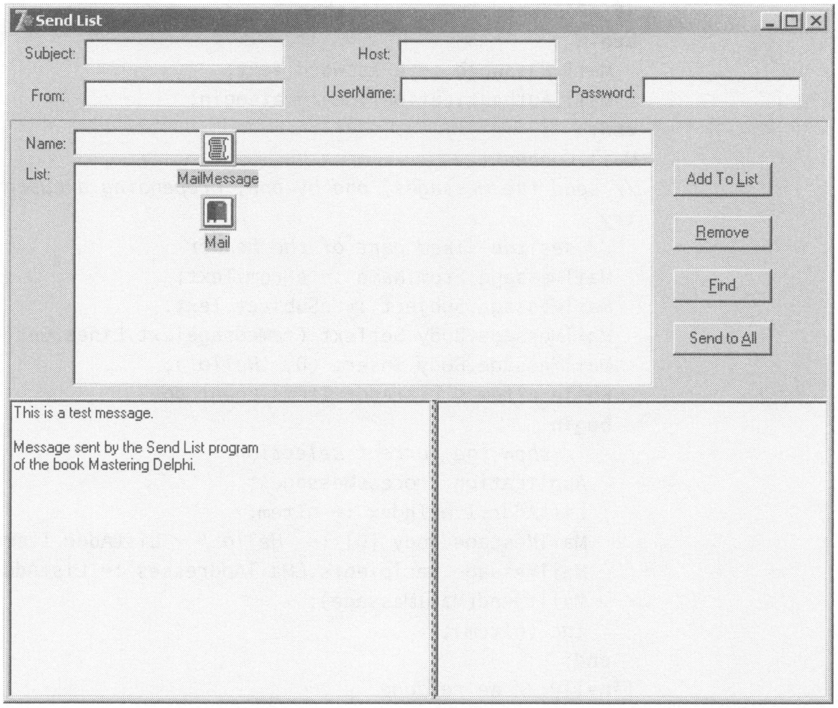
Use of Mail Protocols for Communication with Users Who Are Only Occasionally Online When you must move data between users who are not always online, you can write an application on a server to synchronize among them, and you can give each user a specialized client application for interacting with the server. An alternative is to use an existing server application, such as a mail server, and write the two specialized programs based on the mail protocols. The data sent over this connection will generally be formatted in special ways, so you'll want to use a specific e-mail address for these messages (not your primary e-mail address). As an example, you could rewrite the earlier IndyDbSock example to dispatch mail messages instead of using a custom socket connection. This approach has the advantage of being firewall-friendly and allowing the server to be temporarily offline, because the requests will be kept on the mail server.

Mail In and Out

Using the mail protocols with Indy means placing a message component (IdMessage) in your application, filling it with data, and then using the IdSMTP component to send the mail message. To *retrieve* a mail message from your mailbox, use the IdPop3 component, which will return an IdMessage object. To give you an idea how this process works, I've written a program for sending mail to multiple people at once, using a list stored in an ASCII file. I originally used this program to send mail to people who sign up on my website, but later I extended the program by adding database support and the ability to read subscriber logs automatically. The original version of the program is still a good introduction to the use of the Indy SMTP component.

The SendList program keeps a list of names and e-mail addresses in a local file, which is displayed in a list box. A few buttons allow you to add and remove items, or to modify an item by removing it, editing it, and then adding it again. When the program closes, the updated list is automatically saved. Now let's get to the interesting portion of the program. The top panel, shown in Figure 19.3 at design time, allows you to enter the subject, the sender address, and the information used to connect to the mail server (hostname, username, and eventually a password).

FIGURE 19.3
The SendList program at design time


Send List
Subject:
Host:
From:
UserName:
Password:
Name:
List:
MailMessage
Mail
Add To List
Remove
Find
Send to All

This is a test message.

Message sent by the Send List program
of the book Mastering Delphi.

You'll probably want to make the value of these edit boxes persistent, possibly in an INI file. I haven't done this, only because I don't want you to see my mail connection details! The value of these edit boxes, along with the list of addresses, allows you to send the series of mail messages (after customizing them) with the following code:

```
procedure TMainForm.BtnSendAllClick(Sender: TObject);
var
  nItem: Integer;
  Res: Word;
begin
  Res := MessageDlg ('Start sending from item ' +
    IntToStr (ListAddr.ItemIndex) + ' (' +
    ListAddr.Items [ListAddr.ItemIndex] + ')?'#13 +
    '(No starts from 0)', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
  if Res = mrCancel then
    Exit;
  if Res = mrYes then
    nItem := ListAddr.ItemIndex
  else
    nItem := 0;
  // connect
  Mail.Host := eServer.Text;
  Mail.UserName := eUserName.Text;
```

```
    if ePassword.Text <> '' then
    begin
      Mail.Password := ePassword.Text;
      Mail.AuthenticationType := atLogin;
    end;
    Mail.Connect;
    // send the messages, one by one, prepending a custom message
    try
      // set the fixed part of the header
      MailMessage.From.Name := eFrom.Text;
      MailMessage.Subject := eSubject.Text;
      MailMessage.Body.SetText (reMessageText.Lines.GetText);
      MailMessage.Body.Insert (0, 'Hello');
      while nItem < ListAddr.Items.Count do
      begin
        // show the current selection
        Application.ProcessMessages;
        ListAddr.ItemIndex := nItem;
        MailMessage.Body [0] := 'Hello ' + ListAddr.Items [nItem];
        MailMessage.Recipients.EMailAddresses := ListAddr.Items [nItem];
        Mail.Send(MailMessage);
        Inc (nItem);
      end;
    finally // we're done
      Mail.Disconnect;
    end;
  end;
```

Another interesting example of using mail is to notify developers of problems within applications (a technique you might want to use in an internal application rather than in one you'll distribute widely). You can obtain this effect by modifying the ErrorLog example from Chapter 2 and sending mail when an exception (or one of a given type only) occurs.

Working with HTTP

Handling mail messages is certainly interesting, and mail protocols are probably still the most widespread Internet protocols. The other popular protocol is HTTP, which is used by web servers and web browsers. I'll devote the rest of this chapter to this protocol (along with a discussion of HTML); the following two chapters also discuss it.

On the client side of the Web, the main activity is browsing—reading HTML files. Besides building a custom browser, you can embed the Internet Explorer ActiveX control within your program (as I've done in the WebDemo example in Chapter 12, "From COM to COM+"). You can also directly activate the browser installed on the user's computer—for example, opening an HTML page by calling the `ShellExecute` method (defined in the ShellApi unit):

```
ShellExecute (Handle, 'open', FileName, '', '', sw_ShowNormal);
```

Using `ShellExecute`, you can simply execute a document, such as a file. Windows will start the program associated with the HTM extension, using the action passed as the parameter (in this case, `open`, but passing `nil` would have invoked the standard action producing the same effect). You can use a similar call to view a website, by using a string like 'http://www.example.com' instead of a filename. In this case, the system recognizes the *http* section of the request as requiring a web browser and launches it.

On the server side, you generate and make available the HTML pages. At times, it may be enough to have a way to produce static pages, occasionally extracting new data from a database table to update the HTML files as needed. In other cases, you'll need to generate pages dynamically based on a request from a user.

As a starting point, I'll discuss HTTP by building a simple but complete client and server; then I'll move on to discuss HTML producer components. In Chapter 20, I'll move from this "core technology" level to the RAD development style for the web supported by Delphi, introducing the web server extension technologies (CGI, ISAPI, and Apache modules) and discussing the WebBroker and WebSnap architectures.

Grabbing HTTP Content

As an example of the use of the HTTP protocols, I've written a specific search application. The program hooks onto the Google website, searches for a keyword, and retrieves the first 100 sites found. Instead of showing the resulting HTML file, the program parses it to extract only the URLs of the related sites to a list box. The description of these sites is kept in a separate string list and is displayed as you click a list-box item. So, the program demonstrates two techniques at once: retrieving a web page and parsing its HTML code.

To demonstrate how you should work with blocking connections, such as those used by Indy, I've implemented the program using a background thread for the processing. This approach also gives you the advantage of being able to start multiple searches at once. The thread class used by the WebFind application receives as input a URL to look for, `strUrl`.

The class has two output procedures, `AddToList` and `ShowStatus`, to be called inside the `Synchronize` method. The code of these two methods sends some results or some feedback to the main form, respectively adding a line to the list box and changing the status bar's `SimpleText` property. The key method of the thread is `Execute`. Before we look at it, however, here is how the thread is activated by the main form:

```
const
  strSearch = 'http://www.google.com/search?as_q=';

procedure TForm1.BtnFindClick(Sender: TObject);
var
  FindThread: TFindWebThread;
begin
  // create suspended, set initial values, and start
  FindThread := TFindWebThread.Create (True);
  FindThread.FreeOnTerminate := True;
  // grab the first 100 entries
  FindThread.strUrl := strSearch + EditSearch.Text +'&num=100';
  FindThread.Resume;
end;
```

The URL string is made of the main address of the search engine, followed by some parameters. The first, `as_q`, indicates the words you are looking for. The second, `num=100`, indicates the number of sites to retrieve; you cannot use numbers at will but are limited to a few alternatives, with 100 being the largest possible value.

WARNING *The WebFind program works with the server on the Google website at the time this book was written and tested. The custom software on the site can change, however, which might prevent WebFind from operating correctly. This program was also in* Mastering Delphi 6*; however, it was missing the user agent HTTP header, and after a while Google changed its server software and blocked the requests. Adding any value for the user agent fixed the problem.*

The thread's `Execute` method, activated by the `Resume` call, calls the two methods doing the work (shown in Listing 19.1). In the first, `GrabHtml`, the program connects to the HTTP server using a dynamically created IdHttp component and reads the HTML with the result of the search. The second method, `HtmlToList`, extracts the URLs referring to other websites from the result, the `strRead` string.

LISTING 19.1: THE *TFINDWEBTHREAD* CLASS (OF THE WEBFIND PROGRAM)

```
unit FindTh;

interface

uses
  Classes, IdComponent, SysUtils, IdHTTP;

type
  TFindWebThread = class(TThread)
  protected
    Addr, Text, Status: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (Sender: TObject; AWorkMode: TWorkMode;
      const AWorkCount: Integer);
  public
    strUrl: string;
    strRead: string;
  end;

implementation

{ TFindWebThread }

uses
  WebFindF;
```

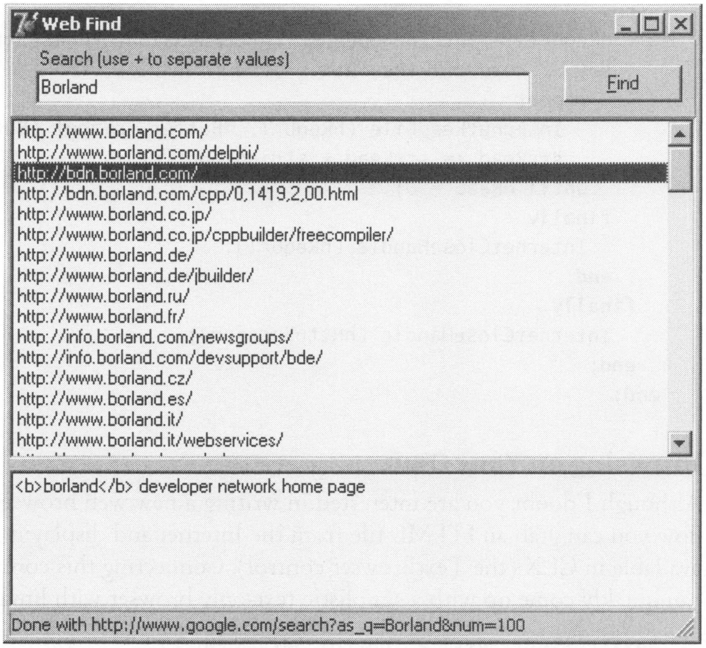
```
procedure TFindWebThread.AddToList;
begin
  if Form1.ListBox1.Items.IndexOf (Addr) < 0 then
  begin
    Form1.ListBox1.Items.Add (Addr);
    Form1.DetailsList.Add (Text);
  end;
end;

procedure TFindWebThread.Execute;
begin
  GrabHtml;
  HtmlToList;
  Status := 'Done with ' + StrUrl;
  Synchronize (ShowStatus);
end;

procedure TFindWebThread.GrabHtml;
var
  Http1: TIdHTTP;
begin
  Status := 'Sending query: ' + StrUrl;
  Synchronize (ShowStatus);
  Http1 := TIdHTTP.Create (nil);
  try
    Http1.Request.UserAgent := 'User-Agent: NULL';
    Http1.OnWork := HttpWork;
    strRead := Http1.Get (StrUrl);
  finally
    Http1.Free;
  end;
end;

procedure TFindWebThread.HtmlToList;
var
  strAddr, strText: string;
  nText: integer;
  nBegin, nEnd: Integer;
begin
  Status := 'Extracting data for: ' + StrUrl;
  Synchronize (ShowStatus);
  strRead := LowerCase (strRead);
  repeat
    // find the initial part HTTP reference
    nBegin := Pos ('href=http', strRead);
    if nBegin <> 0 then
    begin
      // get the remaining part of the string, starting with 'http'
      strRead := Copy (strRead, nBegin + 5, 1000000);
```

```
        // find the end of the HTTP reference
        nEnd := Pos ('>', strRead);
        strAddr := Copy (strRead, 1, nEnd - 1);
        // move on
        strRead := Copy (strRead, nEnd + 1, 1000000);
        // add the URL if 'google' is not in it
        if Pos ('google', strAddr) = 0 then
        begin
          nText := Pos ('</a>', strRead);
          strText := copy (strRead, 1, nText - 1);
          // remove cached references and duplicates
          if (Pos ('cached', strText) = 0) then
          begin
            Addr := strAddr;
            Text := strText;
            AddToList;
          end;
        end;
      end;
    until nBegin = 0;
  end;

  procedure TFindWebThread.HttpWork(Sender: TObject; AWorkMode: TWorkMode;
    const AWorkCount: Integer);
  begin
    Status := 'Received ' + IntToStr (AWorkCount) + ' for ' + strUrl;
    Synchronize (ShowStatus);
  end;

  procedure TFindWebThread.ShowStatus;
  begin
    Form1.StatusBar1.SimpleText := Status;
  end;

  end.
```

The program looks for subsequent occurrences of the `href=http` substring, copying the text up to the closing > character. If the found string contains the word *google*, or its target text includes the word *cached*, it is omitted from the result. You can see the effect of this code in the output shown in Figure 19.4. You can start multiple searches at the same time, but be aware that the results will be added to the same memo component.

THE WININET API

When you need to use the FTP and HTTP protocols, as alternatives to using particular VCL components, you can use a specific API provided by Microsoft in the WinInet DLL. This library is part of the core operating system and implements the FTP and HTTP protocols on top of the Windows sockets API.



Borland

http://www.borland.com/
http://www.borland.com/delphi/
http://bdn.borland.com/
http://bdn.borland.com/cpp/0,1419,2,00.html
http://www.borland.co.jp/
http://www.borland.co.jp/cppbuilder/freecompiler/
http://www.borland.de/
http://www.borland.de/jbuilder/
http://www.borland.ru/
http://www.borland.fr/
http://info.borland.com/newsgroups/
http://info.borland.com/devsupport/bde/
http://www.borland.cz/
http://www.borland.es/
http://www.borland.it/
http://www.borland.it/webservices/

borland developer network home page

Done with http://www.google.com/search?as_q=Borland&num=100

FIGURE 19.4 The WebFind application can be used to search for a list of sites on the Google search engine.

With just three calls—`InternetOpen`, `InternetOpenURL`, and `InternetReadFile`—you can retrieve a file corresponding to any URL and store a local copy or analyze it. Other simple methods can be used for FTP; I suggest you look for the source code of the `WinInet.pas` Delphi unit, which lists all the functions.

TIP *The help file of the WinInet library is not part of the SDK Help shipping with Delphi, but can be found online on MSDN at* `msdn.microsoft.com/library/en-us/wininet/wininet/wininet_reference.asp`.

The `InternetOpen` function establishes a generic connection and returns a handle you can use in the `InternetOpenURL` call. This second call returns a handle to the URL that you can pass to the `InternetReadFile` function in order to read blocks of data. In the following sample code, the data is stored in a local string. When all the data has been read, the program closes the connection to the URL and the Internet session by calling the `InternetCloseHandle` function twice:

```
var
  hHttpSession, hReqUrl: HInternet;
  Buffer: array [0..1023] of Char;
  nRead: Cardinal;
  strRead: string;
  nBegin, nEnd: Integer;
begin
  strRead := '';
  hHttpSession := InternetOpen ('FindWeb', INTERNET_OPEN_TYPE_PRECONFIG,
    nil, nil, 0);
  try
    hReqUrl := InternetOpenURL (hHttpSession, PChar(StrUrl), nil, 0,0,0);
    try // read all the data
      repeat
        InternetReadFile (hReqUrl, @Buffer, sizeof (Buffer), nRead);
        strRead := strRead + string (Buffer);
      until nRead = 0;
    finally
      InternetCloseHandle (hReqUrl);
    end;
  finally
    InternetCloseHandle (hHttpSession);
  end;
end;
```

Browsing on Your Own

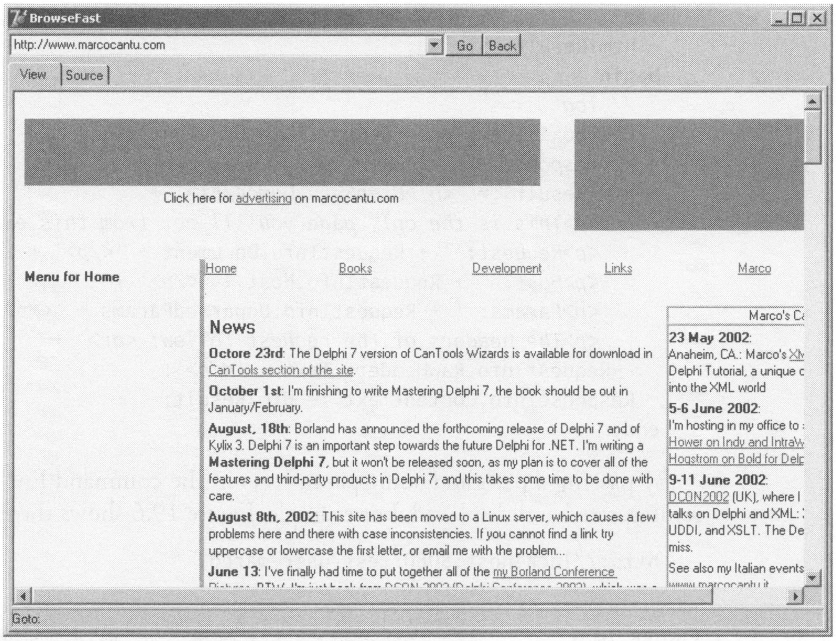
Although I doubt you are interested in writing a new web browser, it might be interesting to see how you can grab an HTML file from the Internet and display it locally, using the HTML viewer available in CLX (the TextBrowser control). Connecting this control to an Indy HTTP client, you can quickly come up with a simplistic text-only browser with limited navigation. The core is

```
TextBrowser1.Text := IdHttp1.Get (NewUrl);
```

where `NewUrl` is complete location of the web resource you want to access. In the BrowseFast example, this URL is entered in a combo box, which keeps track of recent requests. The effect of a similar call is to return the textual portion of a web page (see Figure 19.5), because grabbing the graphic content requires much more complex coding. The TextBrowser control really is better defined as a local file viewer than as a browser.

I've added to the program only very limited support for hyperlinks. When a user moves the mouse over a link, its link text is copied to a local variable (`NewRequest`), which is then used in case of a click on the control to compute the new HTTP request to forward. Merging the current address (`LastUrl`) with the request, though, is far from trivial, even with the help of the `IdUrl` class provided by Indy. Here is my code, which handles only the simplest cases:

```
procedure TForm1.TextBrowser1Click(Sender: TObject);
var
  Uri: TIdUri;
begin
  if NewRequest <> '' then
  begin
    Uri := TIdUri.Create (LastUrl);
    if Pos ('http:', NewRequest) > 0 then
      GoToUrl (NewRequest)
    else if NewRequest [1] = '/' then
      GoToUrl ('http://' + Uri.Host + NewRequest)
    else
      GoToUrl ('http://' + Uri.Host + Uri.Path + NewRequest);
  end;
end;
```



Click here for advertising on marcocantu.com

Menu for Home

News

Octore 23rd: The Delphi 7 version of CanTools Wizards is available for download in CanTools section of the site.

October 1st: I'm finishing to write Mastering Delphi 7, the book should be out in January/February.

August, 18th: Borland has announced the forthcoming release of Delphi 7 and of Kylix 3. Delphi 7 is an important step towards the future Delphi for .NET. I'm writing a **Mastering Delphi 7**, but it won't be released soon, as my plan is to cover all of the features and third-party products in Delphi 7, and this takes some time to be done with care.

August 8th, 2002: This site has been moved to a Linux server, which causes a few problems here and there with case inconsistencies. If you cannot find a link try uppercase or lowercase the first letter, or email me with the problem...

June 13: I've finally had time to put together all of the my Borland Conference

Marco's C

23 May 2002:
Anaheim, CA.: Marco's XM
Delphi Tutorial, a unique c
into the XML world

5-6 June 2002:
I'm hosting in my office to
Hower on Indy and IntraW
Hogstrom on Bold for Delp

9-11 June 2002:
DCON2002 (UK), where I
talks on Delphi and XML:
UDDI, and XSLT. The De
miss.

See also my Italian events

FIGURE 19.5
The output of the BrowseFast text-only browser

Again, this example is trivial and far from usable, but building a browser involves little more than the ability to connect via HTTP and display HTML files.

A Simple HTTP Server

The situation with the development of an HTTP server is quite different. Building a server to deliver static pages based on HTML files is far from simple, although one of the Indy demos provides a good starting point. However, a custom HTTP server might be interesting when building a totally dynamic site, something I'll focus on in more detail in Chapter 20.

To show you how to begin the development of a custom HTTP server, I've built the HttpServ example. This program has a form with a list box used for logging requests and an IdHTTPServer component with these settings:

```
object IdHTTPServer1: TIdHTTPServer
  Active = True
  DefaultPort = 8080
  OnCommandGet = IdHTTPServer1CommandGet
end
```

The server uses port 8080 instead of the standard port 80, so that you can run it alongside another web server. All the custom code is in the `OnCommandGet` event handler, which returns a fixed page plus some information about the request:

```
procedure TForm1.IdHTTPServer1CommandGet(AThread: TIdPeerThread;
  RequestInfo: TIdHTTPRequestInfo; ResponseInfo: TIdHTTPResponseInfo);
```

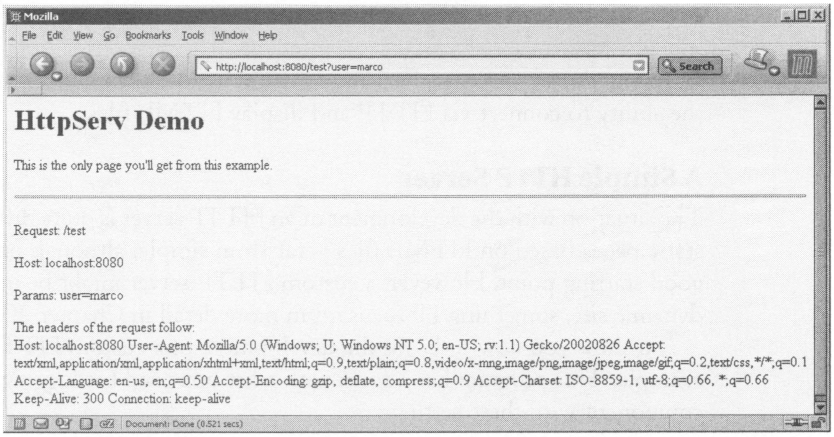
```
var
  HtmlResult: String;
begin
  // log
  Listbox1.Items.Add (RequestInfo.Document);
  // respond
  HtmlResult := '<h1>HttpServ Demo</h1>' +
    '<p>This is the only page you''ll get from this example.</p><hr>' +
    '<p>Request: ' + RequestInfo.Document + '</p>' +
    '<p>Host: ' + RequestInfo.Host + '</p>' +
    '<p>Params: ' + RequestInfo.UnparsedParams + '</p>' +
    '<p>The headers of the request follow: <br>' +
    RequestInfo.RawHeaders.Text + '</p>';
  ResponseInfo.ContentText := HtmlResult;
end;
```

By passing a path and some parameters in the command line of the browser, you'll see them reinterpreted and displayed. For example, Figure 19.6 shows the effect of this command line:

```
http://localhost:8080/test?user=marco
```

FIGURE 19.6

The page displayed by connecting a browser to the custom HttpServ program


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/test?user=marco
Search
Document: Done (0.521 secs)

HttpServ Demo

This is the only page you'll get from this example.

Request: /test

Host: localhost:8080

Params: user=marco

The headers of the request follow:
Host: localhost:8080 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.1) Gecko/20020826 Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/*;q=0.1 Accept-Language: en-us, en;q=0.50 Accept-Encoding: gzip, deflate, compress;q=0.9 Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66 Keep-Alive: 300 Connection: keep-alive

If this example seems too trivial, you'll see a slightly more interesting version in the next section, where I discuss the generation of HTML with Delphi's producer components.

NOTE *If you plan to build an advanced web server or other Internet servers with Delphi, then as an alternative to the Indy components, you should look at the DXSock components from Brain Patchwork DX (*`www.dxsock.com`*).*

Generating HTML

The Hypertext Markup Language, better known by its acronym HTML, is the most widespread format for content on the Web. HTML is the format web browsers typically read; it is a standard defined by the W3C (World Wide Web Consortium, `www.w3.org`), which is one of the bodies controlling the Internet. The HTML standard document is available on `www.w3.org/MarkUp` along with and some interesting links.

Delphi's HTML Producer Components

Delphi's HTML producer components (on the Internet page of the Component Palette) can be used to generate the HTML files and particularly to turn a database table into an HTML table. Many developers believe that the use of these components makes sense only when writing a web server extension. Although they were introduced for this purpose and are part of the WebBroker technology, you can still use three out of the five producer components in any application in which you must generate a static HTML file.

Before looking at the HtmlProd example, which demonstrates the use of these HTML producer components, let me summarize their role:

- The simplest HTML producer component is the PageProducer, which manipulates an HTML file in which you've embedded special tags. The HTML can be stored in an external file or an internal string list. The advantage of this approach is that you can generate such a file using the HTML editor you prefer. At run time, the PageProducer converts the special tags to HTML code, giving you a straightforward method for modifying sections of an HTML document. The special tags have the basic format `<#tagname>`, but you can also supply named parameters within the tag. You'll process the tags in the `OnTag` event handler of the PageProducer.
- The DataSetPageProducer extends the PageProducer by automatically replacing tags corresponding to field names of a connected data source.
- The DataSetTableProducer component is generally useful for displaying the contents of a table, query, or other dataset. The idea is to produce an HTML table from a dataset, in a simple yet flexible way. The component has a nice preview, so you can see how the HTML output will look in a browser directly at design time.
- The QueryTableProducer and the SQLQueryTableProducer components are similar to the DataSetTableProducer, but they are specifically tailored for building parametric queries (for the BDE or dbExpress, respectively) based on input from an HTML search form. This component makes little sense in a stand-alone program, and for this reason, I'll delay covering these components until Chapter 20.

Producing HTML Pages

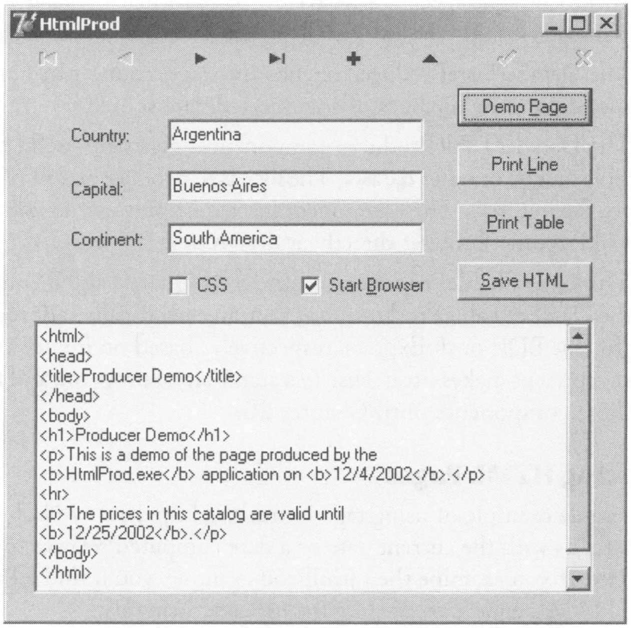
A very simple example of using tags (introduced by the `#` symbol) is creating an HTML file that displays fields with the current date or a date computed relative to the current date, such as an expiration date. If you examine the HtmlProd example, you'll find a PageProducer1 component with internal HTML code, specified by the `HTMLDoc` string list:

```
<html>
<head>
<title>Producer Demo</title>
</head>
<body>
```

```
<h1>Producer Demo</h1>
<p>This is a demo of the page produced by the <b><#appname></b> application on
<b><#date></b>.</p>
<hr>
<p>The prices in this catalog are valid until <b><#expiration
days=21></b>.</p>
</body>
</html>
```

WARNING *If you prepare this file with an HTML editor (something I suggest you do), it may automatically place quotes around tag parameters, as in* `days="21"`*, because this format is required by HTML 4 and XHTML. The PageProducer component has a* `StripParamQuotes` *property you can activate to remove those extra quotes when the component parses the code (before calling the* `OnHTMLTag` *event handler).*

The Demo Page button copies the PageProducer component's output to the `Text` of a Memo. As you call the `Content` function of the PageProducer component, it reads the input HTML code, parses it, and triggers the `OnTag` event handler for every special tag. In the handler for this event, the program checks the value of the tag (passed in the `TagString` parameter) and returns a different HTML text (in the `ReplaceText` reference parameter), producing the output shown in Figure 19.7.



Country: Argentina
Capital: Buenos Aires
Continent: South America
☐ CSS ☑ Start Browser

```
<html>
<head>
<title>Producer Demo</title>
</head>
<body>
<h1>Producer Demo</h1>
<p>This is a demo of the page produced by the
<b>HtmlProd.exe</b> application on <b>12/4/2002</b>.</p>
<hr>
<p>The prices in this catalog are valid until
<b>12/25/2002</b>.</p>
</body>
</html>
```

FIGURE 19.7 The output of the HtmlProd example, a simple demonstration of the PageProducer component, when the user clicks the Demo Page button

```
procedure TFormProd.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
```

```
var
  nDays: Integer;
begin
  if TagString = 'date' then
    ReplaceText := DateToStr (Now)
  else if TagString = 'appname' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'expiration' then
  begin
    nDays := StrToIntDef (TagParams.Values['days'], 0);
    if nDays <> 0 then
      ReplaceText := DateToStr (Now + nDays)
    else
      ReplaceText := '<i>{expiration tag error}</i>';
  end;
end;
```

Notice in particular the code I've written to convert the last tag, `#expiration`, which requires a parameter. The PageProducer places the entire text of the tag parameter (in this case, *days=21*) in a string that's part of the `TagParams` list. To extract the value portion of this string (the portion after the equal sign), you can use the `Values` property of the `TagParams` string list and search for the proper entry at the same time. If it can't locate the parameter or if the parameter's value isn't an integer, the program displays an error message.

TIP The PageProducer component supports user-defined tags, which can be any string you like, but you should first review the special tags defined by the `TTags` *enumeration. The possible values include tgLink (for the* `link` *tag), tgImage (for the* `img` *tag), tgTable (for the* `table` *tag), and a few others. If you create a custom tag, as in the PageProd example, the value of the* `Tag` *parameter to the HTMLTag handler will be tgCustom.*

Producing Pages of Data

The HtmlProd example also has a DataSetPageProducer component, which is connected with a database table and with the following HTML source code:

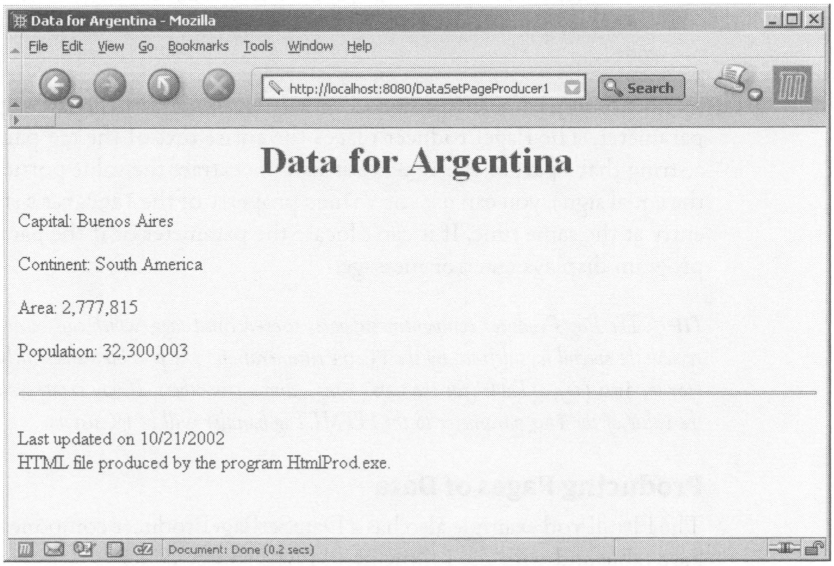
```
<html><head><title>Data for <#name></title></head>
<body>
<h1><center>Data for <#name></center></h1>
<p>Capital: <#capital></p>
<p>Continent: <#continent></p>
<p>Area: <#area></p>
<p>Population: <#population></p><hr>
<p>Last updated on <#date><br>
HTML file produced by the program <#program>.</p>
</body></html>
```

By using tags with the names of the connected dataset's fields (the usual COUNTRY.DB database table), the program automatically gets the value of the current record's fields and replaces them automatically. This produces the output shown in Figure 19.8; the browser is connected to the HtmlProd example working as an HTTP server, as I'll discuss later. In the source code of the program related to this component, there is no reference to the database data:

```
procedure TFormProd.DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'program' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'date' then
    ReplaceText := DateToStr (Date);
end;
```

FIGURE 19.8

The output of the HtmlProd example for the Print Line button


Data for Argentina - Mozilla
File
Edit
View
Go
Bookmarks
Tools
Window
Help
http://localhost:8080/DataSetPageProducer1
Search
Document: Done (0.2 secs)

Data for Argentina

Capital: Buenos Aires

Continent: South America

Area: 2,777,815

Population: 32,300,003

Last updated on 10/21/2002
HTML file produced by the program HtmlProd.exe.

Producing HTML Tables

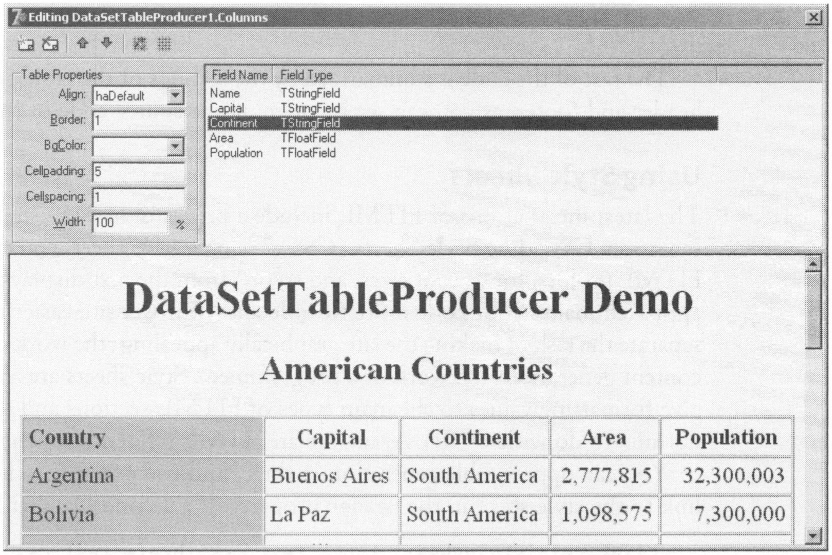
The third button in the HtmlProd example is Print Table. This button is connected to a DataSetTableProducer component, again calling its `Content` function and copying its result to the `Text` of the Memo. By connecting the `DataSet` property of the DataSetTableProducer to `ClientDataSet1`, you can produce a standard HTML table.

The component by default generates only 20 rows, as indicated by the `MaxRows` property. If you want to get all of the table's records, you can set this property to -1—a simple but undocumented setting.

TIP *The DataSetTableProducer component starts from the current record rather than from the first one. So, the second time you click the Print Table button, you'll see no records in the output. Adding a call to the dataset's* `First` *method before calling the producer component's* `Content` *method fixes the problem.*

To make the output of this producer component more complete, you can perform two operations. The first is to provide some `Header` and `Footer` information (to generate the HTML heading and closing elements) and add a `Caption` to the HTML table. The second is to customize the table by using the setting specified by the `RowAttributes`, `TableAttributes`, and `Columns` properties. The property editor for the columns, which is also the default component editor, allows you to set most of these properties, providing at the same time a nice preview of the output, as you can see in Figure 19.9. Before using this editor, you can set up properties for the dataset's fields using the Fields editor. This is how, for example, you can format the output of the population and area fields to use thousands separators.

FIGURE 19.9
The editor of the DataSetTableProducer component's `Columns` property provides you with a preview of the final HTML table (if the database table is active).



Align: haDefault
Border: 1
BgColor:
Cellpadding: 5
Cellspacing: 1
Width: 100 %

Field Name	Field Type
Name	TStringField
Capital	TStringField
Continent	TStringField
Area	TFloatField
Population	TFloatField

DataSetTableProducer Demo

American Countries

Country	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2,777,815	32,300,003
Bolivia	La Paz	South America	1,098,575	7,300,000

You can use three techniques to customize the HTML table, and it's worth reviewing each of them:

- You can use the table producer component's `Column's` property to set properties, such as the text and color of the title, or the color and the alignment for the cells in the rest of the column.
- You can use the `TField` properties, particularly those related to output. In the example, I've set the `DisplayFormat` property of the `ClientDataSet1Area` field object to ###,###,###. This is the approach to use if you want to determine the output of each field. You might go even further and embed HTML tags in the output of a field.
- You can handle the DataSetTableProducer component's `OnFormatCell` event to customize the output further. In this event, you can set the various column attributes uniquely for a given cell, but you can also customize the output string (stored in the `CellData` parameter) and embed HTML tags. You can't do this using the `Columns` property.

In the HtmlProd example, I've used a handler for this event to turn the text of the Population and Area columns to bold font and to a red background for large values (unless it is the header row). Here is the code:

```
procedure TFormProd.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
```

```
    var BgColor: THTMLBgColor; var Align: THTMLAlign;
    var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
  begin
    if (CellRow > 0) and
      (((CellColumn = 3) and (Length (CellData) > 8)) or
      ((CellColumn = 4) and (Length (CellData) > 9))) then
    begin
      BgColor := 'red';
      CellData := '<b>' + CellData + '</b>';
    end;
  end;
```

The rest of the code is summarized by the settings of the table producer component, including its header and footer, as you can see by opening the source code of the HtmlProd example.

Using Style Sheets

The latest incarnations of HTML include a powerful mechanism for separating content from presentation: Cascading Style Sheets (CSS). Using a style sheet, you can separate the formatting of the HTML (colors, fonts, font sizes, and so on) from the text displayed (the content of the page). This approach makes your code more flexible and your website easier to update. In addition, you can separate the task of making the site graphically appealing (the work of a web designer) from automatic content generation (the work of a programmer). Style sheets are a complex technique, in which you give formatting values to the main types of HTML sections and to special "classes" (which have nothing to do with OOP). Again, see an HTML reference for the details.

You can update table generation in the HtmlProd example to include style sheets by providing a link to the style sheet in the `Header` property of a second DataSetTableProducer component:

```
<link rel="stylesheet" type="text/css" href="test.css">
```

You can then update the code of the `OnFormatCell` event handler with the following action (instead of the two lines changing the color and adding the bold font tag):

```
CustomAttrs := 'class="highlight"';
```

The style sheet I've provided (`test.css`, available in the source code of the example) defines a highlight style, which has the bold font and red background that were hard-coded in the first DataSetTableProducer component.

The advantage of this approach is that now a graphic artist can modify the CSS file and give your table a nicer look without touching its code. When you want to provide many formatting elements, using a style sheet can also reduce the total size of the HTML file. This is an important element that can reduce download time.

Dynamic Pages from a Custom Server

The HtmlProd component can be used to generate static HTML files; it doubles as a web server, using an approach similar to that demonstrated in the HttpServ example, but in a more realistic context. The program accesses the request of one of the possible page producers, passing the name of the component in a request. This is a portion of the IdHTTPServer component's `OnCommandGet` event handler, which uses the `FindComponent` method to locate the proper producer component:

```
var
  Req, Html: String;
  Comp: TComponent;
begin
  Req := RequestInfo.Document;
  if Req [1] = '/' then
    Req := Copy (Req, 2, 1000); // skip '/'
  Comp := FindComponent (Req);
  if (Req <> '') and Assigned (Comp) and
    (Comp is TCustomContentProducer) then
  begin
    ClientDataSet1.First;
    Html := TCustomContentProducer (Comp).Content;
  end;
  ResponseInfo.ContentText := Html;
end;
```

In case the parameter is not there (or is not valid), the server responds with an HTML-based menu of the available components:

```
Html := '<h1>HtmlProd Menu<h1><p><ul>';
for I := 0 to ComponentCount - 1 do
  if Components [i] is TCustomContentProducer then
    Html := Html + '<li><a href="/' + Components [i].Name + '">' +
      Components [i].Name + '</a></li>';
Html := Html + '</ul></p>';
```

Finally, if the program returns a table that uses CSS, the browser will request the CSS file from the server; so, I've added some specific code to return it. With the proper generalizations, this code shows how a server can respond by returning files, and also how to indicate the MIME type of the response (`ContentType`):

```
if Pos ('test.css', Req) > 0 then
begin
  CssTest := TStringList.Create;
  try
    CssTest.LoadFromFile(ExtractFilePath(Application.ExeName) + 'test.css');
    ResponseInfo.ContentText := CssTest.Text;
    ResponseInfo.ContentType := 'text/css';
  finally
    CssTest.Free;
  end;
  Exit;
end;
```

What's Next?

In this chapter, I've focused on some core Internet technologies, including the use of sockets and core Internet protocols. I've discussed the main idea and shown a few examples of using the mail and HTTP protocols. You can find many more examples that use the Indy components in the demos written by their developers (which are not installed in Delphi 7).

After this introduction to the world of the Internet, you are now ready to delve into two key areas: the present and the future. The present is represented by the development of web applications, and I'll explore the development of dynamic websites in the next two chapters. I'll focus first on the old WebBroker technology and then move to the new WebSnap architecture. Then, in Chapter 21, I'll discuss IntraWeb. The future is represented by the development of web services and the use of XML and related technology, discussed in Chapters 22 and 23.

Chapter 20

Web Programming with WebBroker and WebSnap

The Internet has a growing role in the world, and much of it depends on the success of the World Wide Web, which is based on the HTTP protocol. In Chapter 19, "Internet Programming: Sockets and Indy," we discussed HTTP and the development of client- and server-side applications based on it. With the availability of several high-performance, scalable, flexible web servers, you'll rarely want to create your own. Dynamic web server applications are generally built by integrating scripting or compiled programs within web servers, rather than by replacing them with custom software.

This chapter is entirely focused on the development of server-side applications, which extend existing web servers. I introduced the dynamic generation of HTML pages toward the end of the last chapter. Now you will learn how to integrate this dynamic generation within a server. This chapter is a logical continuation of the last one, but it won't complete this book's coverage of Internet programming; Chapter 21 is devoted to the IntraWeb technology available in Delphi 7, and Chapter 22 gets back to Internet programming from the XML perspective.

WARNING *To test some of the examples in this chapter, you'll need access to a web server. The simplest solution is to use the version of Microsoft's IIS or Personal Web Server already installed on your computer. My personal preference, however, is to use the free open-source Apache Web Server available at* `www.apache.org`*. I won't spend much time giving you details of the configuration of your web server to enable the use of applications; you can refer to its documentation for this information.*

This chapter covers the following:

- Dynamic web pages
- CGI, ISAPI, and Apache modules
- The WebBroker architecture
- The Web App Debugger
- The WebSnap architecture
- Adapters and server-side scripting

Dynamic Web Pages

When you browse a website, you generally download static pages—HTML-format text files—from the web server to your client computer. As a web developer, you can create these pages manually, but for most businesses, it makes more sense to build the static pages from information in a database (a SQL server, a series of files, and so on). Using this approach, you're basically generating a snapshot of the data in HTML format, which is reasonable if the data isn't subject to frequent changes. This approach was discussed in Chapter 19.

As an alternative to static HTML pages, you can build dynamic pages. To do this, you extract information directly from a database in response to the browser's request, so that the HTML sent by your application displays current data, not an old snapshot of the data. This approach makes sense if the data changes frequently.

As mentioned earlier, there are a couple of ways you can program custom behavior at the web server, and these are ideal techniques you can use to generate HTML pages dynamically. In addition to script-based techniques, which are very popular, two common protocols for programming web servers are CGI (the Common Gateway Interface) and the web server APIs.

NOTE *Keep in mind that Delphi's WebBroker technology (available in both the Enterprise Studio and Professional editions) flattens the differences between CGI and server APIs by providing a common class framework. This way, you can easily turn a CGI application into an ISAPI library or integrate it into Apache.*

An Overview of CGI

CGI is a standard protocol for communication between the client browser and the web server. It's not a particularly efficient protocol, but it is widely used and is not platform specific. This protocol allows the browser both to ask for and to send data, and it is based on the standard command-line input and output of an application (usually a console application). When the server detects a page request for the CGI application, it launches the application, passes command-line data from the page request to the application, and then sends the application's standard output back to the client computer.

You can use many tools and languages to write CGI applications, and Delphi is only one of them. Despite the obvious limitation that your web server must be an Intel-based Windows or Linux system, you can build some fairly sophisticated CGI programs in Delphi and Kylix. CGI is a low-level technique, because it uses the standard command-line input and output along with environment variables to receive information from the web server and pass it back.

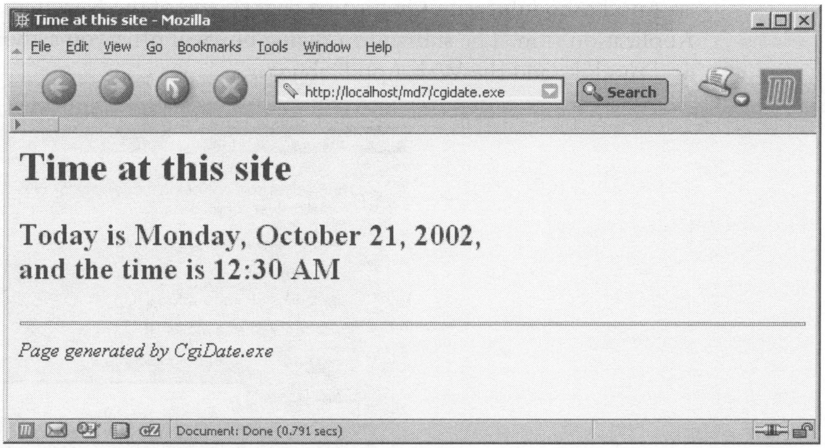
To build a CGI program without using support classes, you can create a Delphi console application, remove the typical project source code, and replace it with the following statements:

```
program CgiDate;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
  writeln ('content-type: text/html');
  writeln;
  writeln ('<html><head>');
```

```
    writeln ('<title>Time at this site</title>');
    writeln ('</head><body>');
    writeln ('<h1>Time at this site</h1>');
    writeln ('<hr>');
    writeln ('<h3>');
    writeln (FormatDateTime('"Today is " dddd, mmmm d, yyyy,' +
      '"<br> and the time is" hh:mm:ss AM/PM', Now));
    writeln ('</h3>');
    writeln ('<hr>');
    writeln ('<i>Page generated by CgiDate.exe</i>');
    writeln ('</body></html>');
  end.
```

CGI programs produce a header followed by the HTML text using the standard output. If you execute this program directly, you'll see the text in a terminal window. If you run it instead from a web server and send the output to a browser, the formatted HTML text will appear, as shown in Figure 20.1.

FIGURE 20.1
The output of the CgiDate application, as seen in a browser



Time at this site - Mozilla

http://localhost/md7/cgidate.exe

Time at this site

Today is Monday, October 21, 2002,
and the time is 12:30 AM

Page generated by CgiDate.exe

Document: Done (0.791 secs)

Building advanced applications with plain CGI requires a lot of work. For example, to extract status information about the HTTP request, you need to access the relevant environment variables, as in the following:

```
// get the pathname
GetEnvironmentVariable ('PATH_INFO', PathName, sizeof (PathName));
```

Using Dynamic Libraries

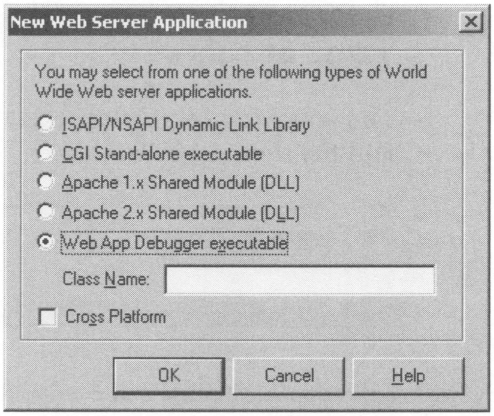
A completely different approach is the use of the web server APIs: the popular ISAPI (Internet Server API, introduced by Microsoft), the less common NSAPI (Netscape Server API), or the Apache API. These APIs allow you to write a library that the server loads into its own address space and keeps in memory. Once it loads the library, the server can execute individual requests via threads within the main process, instead of launching a new EXE for every request (as it must in CGI applications).

When the server receives a page request, it loads the DLL (if it hasn't done so already) and executes the appropriate code, which may launch a new thread or use an existing one to process the request. The library then sends the appropriate HTTP data back to the client that requested the page. Because this communication generally occurs in memory, this type of application tends to be faster than CGI.

Delphi's WebBroker Technology

The CGI code snippet I've shown you demonstrates the plain, direct approach to this protocol. I could have provided similar low-level examples for ISAPI or Apache modules, but in Delphi it's more interesting to use the WebBroker technology. This comprises a class hierarchy within VCL and CLX (built to simplify server-side development on the Web) and a specific type of data modules called *WebModules*. Both the Enterprise Studio and Professional editions of Delphi include this framework (as opposed to the more advanced and newer WebSnap framework, which is available only in the Enterprise Studio version).

Using the WebBroker technology, you can begin developing an ISAPI or CGI application or an Apache module easily. On the first page (New) of the New Items dialog box, select the Web Server Application icon. The subsequent dialog box will offer you a choice among ISAPI, CGI, Apache 1 or 2 module, and the Web App Debugger:



You may select from one of the following types of World Wide Web server applications.

○ ISAPI/NSAPI Dynamic Link Library
○ CGI Stand-alone executable
○ Apache 1.x Shared Module (DLL)
○ Apache 2.x Shared Module (DLL)
● Web App Debugger executable

Class Name:

☐ Cross Platform

In each case, Delphi will generate a project with a WebModule, which is a non-visual container similar to a data module. This unit will be identical, regardless of the project type; only the main project file changes. For a CGI application, it will look like this:

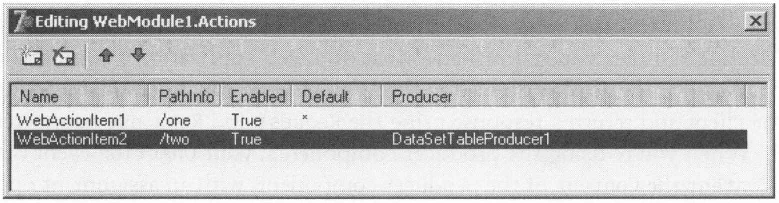
```
program Project2;

{$APPTYPE CONSOLE}

uses
  WebBroker,
  CGIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};
```

```
{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Although this is a console CGI program, the code looks similar to that of a standard Delphi application. However, it uses a trick—the `Application` object used by this program is not the typical global object of class `TApplication` but an object of a new class. This `Application` object is of class `TCGIApplication` or another class derived from `TWebApplication`, depending on your web project type.

The most important operations take place in the WebModule. This component derives from `TCustomWebDispatcher`, which provides support for all the input and output of your programs. The `TCustomWebDispatcher` class defines `Request` and `Response` properties, which store the client request and the response you're going to send back to the client. Each of these properties is defined using a base abstract class (`TWebRequest` and `TWebResponse`), but an application initializes them using a specific object (such as the `TISAPIRequest` and `TISAPIResponse` subclasses). These classes make available all the information passed to the server, so you have a single approach to accessing all the information. The same is true of a response, which is easy to manipulate. The key advantage of this approach is that the code written with WebBroker is independent of the type of application (CGI, ISAPI, Apache module); you'll be able to move from one to the other, modifying the project file or switching to another one, but you won't need to modify the code written in a WebModule.

This is the structure of Delphi's framework. To write the application code, you can use the Actions editor in the WebModule to define a series of actions (stored in the `Actions` array property) depending on the *pathname* of the request:



Name	PathInfo	Enabled	Default	Producer
WebActionItem1	/one	True	*	
WebActionItem2	/two	True		DataSetTableProducer1

This pathname is a portion of the CGI or ISAPI application's URL, which comes after the program name and before the parameters, such as `path1` in the following URL:

```
http://www.example.com/scripts/cgitest.exe/path1?param1=date
```

By providing different actions, your application can easily respond to requests with different pathnames, and you can assign a different producer component or call a different `OnAction` event handler for every possible pathname. Of course, you can omit the pathname to handle a generic request. Also consider that instead of basing your application on a WebModule, you can use a plain data module and add a WebDispatcher component to it. This is a good approach if you want to turn an existing Delphi application into a web server extension.

WARNING *The WebModule inherits from the base WebDispatcher class and doesn't require it as a separate component. Unlike WebSnap applications, WebBroker programs cannot have multiple dispatchers or multiple web modules. Also note that the actions of the WebDispatcher have nothing to do with the actions stored in an ActionList or ActionManager component.*

When you define the accompanying HTML pages that launch the application, the links will make page requests to the URLs for each of those paths. Having a single library that can perform different operations depending on a parameter (in this case, the pathname) allows the server to keep a copy of this library in memory and respond much more quickly to user requests. The same is partially true for a CGI application: The server has to run several instances but can cache the file and make it available more quickly.

In the `OnAction` event, you write the code to specify the *response* to a given *request*, the two main parameters passed to the event handler. Here is an example:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    '<html><head><title>Hello Page</title></head><body>' +
    '<h1>Hello</h1>' +
    '<hr><p><i>Page generated by Marco</i></p></body></html>';
end;
```

In the `Content` property of the `Response` parameter, you enter the HTML code you want users to see. The only drawback of this code is that the output in a browser will be correctly displayed on multiple lines, but looking at the HTML source code, you'll see a single line corresponding to the entire string. To make the HTML source code more readable by splitting it onto multiple lines, you can insert the `#13` newline character (or, even better, the cross-platform `sLineBreak` value).

To let other actions handle this request, you set the last parameter, `Handled`, to False. The default value is True; if this value is set, then once you've handled the request with your action, the WebModule assumes you're finished. Most of a web application's code will be in the `OnAction` event handlers for the actions defined in the WebModule container. These actions receive a request from the client and return a response using the `Request` and `Response` parameters.

When you're using the producer components, your `OnAction` event often returns as `Response.Content` the `Content` of the producer component, with an assignment operation. You can shortcut this code by assigning a producer component to the `Producer` property of the action, and you won't need to write these event handlers any more (but don't do both things, because doing so might get you into trouble).

TIP *As an alternative to the* `Producer` *property, you can use the* `ProducerContent` *property. This property allows you to connect custom producer classes that don't inherit from the* `TCustomContentProducer` *class but implement the* `IProduceContent` *interface. The* `ProducerContent` *property is* almost *an interface property: It behaves the same way, but this behavior is due to its property editor and is not based on Delphi's support for interfaced properties.*

Debugging with the Web App Debugger

Debugging web applications written in Delphi is often difficult. You cannot simply run the program and set breakpoints in it, but must convince the web server to run your CGI program or library within the Delphi debugger. You can do so by indicating a host application in Delphi's Run Parameters dialog box, but this approach implies letting Delphi run the web server (which is often a Windows service, not a stand-alone program).

To solve these issues, Borland has developed a specific Web App Debugger program. This program, which is activated by the corresponding item on the Tools menu, is a web server that waits for requests on a port you can set up (1024 by default). When a request arrives, the program can forward it to a stand-alone executable. In Delphi 6, this communication was based on COM techniques; in Delphi 7 it is based on Indy sockets. In both cases, you can run the web server application from within the Delphi IDE, set all the breakpoints you need, and then (when the program is activated through the Web App Debugger) debug the program as you would a plain executable file.

The Web App Debugger does a good job of logging all the received requests and the responses returned to the browser. The program also has a Statistics page that tracks the time required for each response, allowing you to test the efficiency of an application in different conditions. Another new feature of the Web App Debugger in Delphi 7 is that it is now a CLX application instead of a VCL application. This user interface change and the conversion from COM to sockets were both done to make the Web App Debugger available in Kylix.

WARNING *Because the Web App Debugger uses Indy sockets, your application will receive frequent exceptions of type* `EidConnClosedGracefully`. *For this reason, this exception is automatically disabled in all Delphi 7 projects.*

By using the corresponding option in the New Web Server Application dialog, you can easily create a new application compatible with the debugger. This option defines a standard project, which creates both a main form and a web module. The (useless) form includes code for providing initialization code and adding the application to the Windows Registry:

```
initialization
  TWebAppSockObjectFactory.Create('program_name');
```

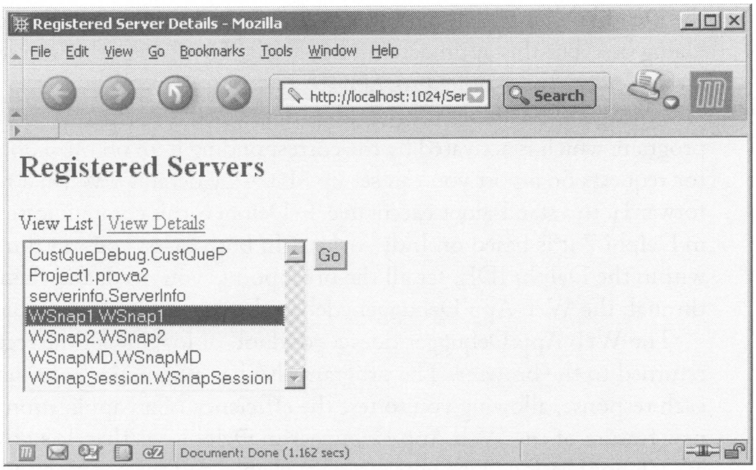
The Web App Debugger uses this information to get a list of the available programs. It does so when you use the default URL for the debugger, indicated in the form as a link, as you can see (for example) in Figure 20.2. The list includes all the registered servers, not only those that are running, and can be used to activate a program. This is not a good idea, though, because you have to run the program within the Delphi IDE to be able to debug it. (Notice that you can expand the list by clicking View Details; this view includes a list of the executable files and many other details.)

The data module for this type of project includes initialization code:

```
uses WebReq;

initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.WebModuleClass := TWebModule2;
```

The Web App Debugger should be used only for debugging. To deploy the application, you should use one of the other options. You can create the project files for another type of web server program and add to the project the same web module as the debug application.

The reverse process is slightly more complex. To debug an existing application, you have to create a program of this type, remove the web module, add the existing one, and patch it by adding a line to set the `WebModuleClass` of the `WebRequestHandler`, as in the preceding code snippet.


Registered Server Details - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:1024/Ser
Search
Go
Document: Done (1.162 secs)

Registered Servers

View List | View Details

CustQueDebug.CustQueP
Project1.prova2
serverinfo.ServerInfo
WSnap1.WSnap1
WSnap2.WSnap2
WSnapMD.WSnapMD
WSnapSession.WSnapSession

FIGURE 20.2
A list of applications registered with the Web App Debugger is displayed when you hook to its home page.

WARNING *Although in most cases you'll be able to move a program from one web technology to another, this is not always the case. For example, in the CustQueP example (discussed later), I had to avoid the request's* `ScriptName` *property (which is fine for a CGI program) and use the* `InternalScriptName` *property instead.*

There are two other interesting elements involved in using the Web App Debugger. First, you can test your program without having a web server installed and without having to tweak its settings. In other words, you don't have to deploy your programs to test them—you can try them right away. Second, instead of doing early development of an application as CGI, you can begin experimenting with a multithreaded architecture immediately, without having to deal with the loading and unloading of libraries (which often implies shutting down the web server and possibly even the computer).

Building a Multipurpose WebModule

To demonstrate how easily you can build a feature-rich server-side application using Delphi's support, I created the BrokDemo example. I built this example using the Web App Debugger technology, but it should be relatively simple to recompile as a CGI or a web server library.

A key element of the WebBroker example is the list of actions. The actions can be managed in the Actions editor or directly in the Object TreeView. Actions are also visible in the Designer page of the editor, so you can graphically see their relationship with database objects. If you examine the source code, you'll notice that every action has a specific name. I also gave meaningful names to the `OnAction` event handlers. For instance, `TimeAction` as a method name is much more understandable than the `WebModule1WebActionItem1Action` name automatically generated by Delphi.

Every action has a different pathname, and one is marked as a default and executed even if no pathname is specified. The first interesting idea in this program is the use of two PageProducer components, `PageHead` and `PageTail`, which are used for the initial and final portion of every page. Centralizing this code makes it easier to modify, particularly if it is based on external HTML files. The HTML produced by these components is added at the beginning and the end of the resulting HTML in the web module's `OnAfterDispatch` event handler:

```
procedure TWebModule1.WebModule1AfterDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageHead.Content + Response.Content + PageTail.Content;
end;
```

You add the initial and final HTML at the end of the page generation because doing so allows the components to produce the HTML as if they were making all of it. Starting with HTML in the `OnBeforeDispatch` event means that you cannot directly assign the producer components to the actions, or the producer component will override the `Content` you've already provided in the response.

The `PageTail` component includes a custom tag for the script name, replaced by the following code, which uses the current request object available in the web module:

```
procedure TWebModule1.PageTailHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'script' then
    ReplaceText := Request.ScriptName;
end;
```

This code is activated to expand the `<#script>` tag of the `PageTail` component's `HTMLDoc` property.

The code for the time and date actions is straightforward. The really interesting part begins with the Menu path, which is the default action. In its `OnAction` event handler, the application uses a `for` loop to build a list of the available actions (using their names without the first two letters, which are always `Wa` in this example), providing a link to each of them with an anchor (an `<a>` tag):

```
procedure TWebModule1.MenuAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<h3>Menu</h3><ul>'#13;
  for I := 0 to Actions.Count - 1 do
    Response.Content := Response.Content + '<li> <a href="' +
      Request.ScriptName + Action[I].PathInfo + '"> ' +
      Copy (Action[I].Name, 3, 1000) + '</a>'#13;
  Response.Content := Response.Content + '</ul>';
end;
```

The BrokDemo example also provides users with a list of the system settings related to the request, which is useful for debugging. It is also instructive to learn how much information (and exactly what information) the HTTP protocol transfers from a browser to a web server and vice versa. To produce this list, the program looks for the value of each property of the `TWebRequest` class, as this snippet demonstrates:

```
procedure TWebModule1.StatusAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
```

```
begin
  Response.Content := '<h3>Status</h3>'#13 +
    'Method: ' + Request.Method + '<br>'#13 +
    'ProtocolVersion: ' + Request.ProtocolVersion + '<br>'#13 +
    'URL: ' + Request.URL + '<br>'#13 +
    'Query: ' + Request.Query + '<br>'#13 + ...
```

Dynamic Database Reporting

The BrokDemo example defines two more actions, indicated by the `/table` and `/record` pathnames. For these two last actions, the program produces a list of names and then displays the details of one record, using a DataSetTableProducer component to format the entire table and a DataSetPageProducer component to build the record view. Here are the properties of these two components:

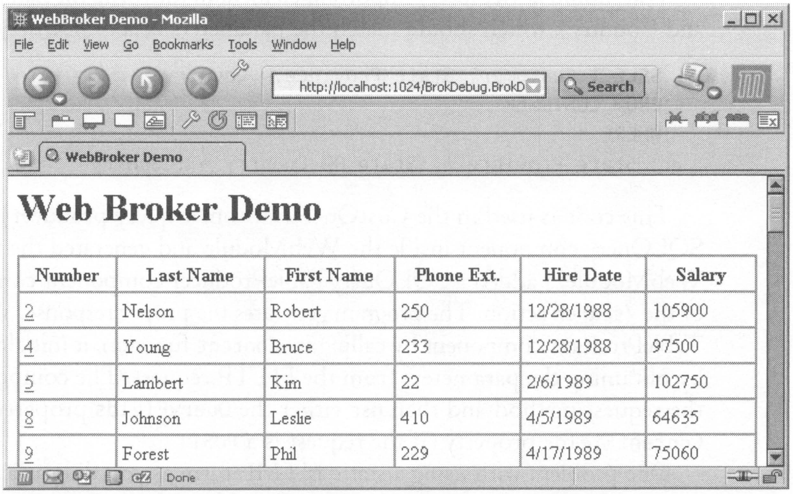
```
object DataSetTableProducer1: TDataSetTableProducer
  DataSet = dataEmployee
  OnFormatCell = DataSetTableProducer1FormatCell
end
object DataSetPage: TDataSetPageProducer
  HTMLDoc.Strings = (
    '<h3>Employee: <#LastName></h3>'
    '<ul><li> Employee ID: <#EmpNo>'
    '<li> Name: <#FirstName> <#LastName>'
    '<li> Phone: <#PhoneExt>'
    '<li> Hired On: <#HireDate>'
    '<li> Salary: <#Salary></ul>')
  OnHTMLTag = PageTailHTMLTag
  DataSet = dataEmployee
end
```

To produce the entire table, you connect the DataSetTableProducer to the `Producer` property of the corresponding actions without providing a specific event handler. The table is made more powerful by adding internal links to the specific records. The following code is executed for each cell of the table but a link is created only for the first column and not for the first row (the one with the title):

```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign;
  var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + ScriptName + '/record?LastName=' +
      dataEmployee['Last_Name'] + '&FirstName=' +
      dataEmployee['First_Name'] + '"> '
      + CellData + ' </a>';
end;
```

You can see the result of this action in Figure 20.3. When the user selects one of the links, the program is called again, and it can check the `QueryFields` string list and extract the parameters from the URL. It then uses the values corresponding to the table fields used for the record search (which is based on the `FindNearest` call):

```
procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  dataEmployee.Open;
  // go to the requested record
  dataEmployee.Locate ('LAST_NAME;FIRST_NAME',
    VarArrayOf([Request.QueryFields.Values['LastName'],
    Request.QueryFields.Values['FirstName']]), []);
  // get the output
  Response.Content := Response.Content + DataSetPage.Content;
end;
```


WebBroker Demo - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:1024/BrokDebug.BrokD
Search
WebBroker Demo
Done

Web Broker Demo

Number	Last Name	First Name	Phone Ext.	Hire Date	Salary
2	Nelson	Robert	250	12/28/1988	105900
4	Young	Bruce	233	12/28/1988	97500
5	Lambert	Kim	22	2/6/1989	102750
8	Johnson	Leslie	410	4/5/1989	64635
9	Forest	Phil	229	4/17/1989	75060

FIGURE 20.3
The output corresponding to the table path of the BrokDemo example, which produces an HTML table with internal hyperlinks

Queries and Forms

The previous example used some of the HTML producer components introduced earlier in this chapter. This group includes another component you haven't used yet: QueryTableProducer (for the BDE) and its sister SQL QueryTableProducer (for dbExpress). As you'll see in a moment, this component makes building even complex database programs a breeze.

Suppose you want to search for customers in a database. You might construct the following HTML form (embedded in an HTML table for better formatting):

```
<h4>Customer QueryProducer Search Form</h4>
<form action="<#script>/search" method="POST">
```

```
<table>
<tr><td>State:</td>
  <td><input type="text" name="State"></td></tr>
<tr><td>Country:</td>
  <td><input type="text" name="Country"></td></tr>
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

NOTE *As in Delphi, an HTML form hosts a series of controls. There are visual tools to help you design these forms, or you can manually enter the HTML code. The available controls include buttons, input text (or edit boxes), selections (or combo boxes), and input buttons (or radio buttons). You can define buttons as specific types, such as Submit or Reset, which imply standard behaviors. An important element of forms is the* request method, *which can be either* `POST` *(data is passed behind the scenes, and you receive it in the* `ContentFields` *property) or* `GET` *(data is passed as part of the URL, and you extract it from the* `QueryFields` *property).*

You should notice a very important element in the form: the names of the input components (State and Country), which should match the parameters of a SQLQuery component:

```
SELECT Customer, State_Province,  Country
FROM CUSTOMER
WHERE
  State_Province = :State OR Country = :Country
```

This code is used in the CustQueP (customer query producer) example. To build it, I placed a SQLQuery component inside the WebModule and generated the field objects for it. In the same WebModule, I added a SQLQueryTableProducer component connected to the `Producer` property of the `/search` action. The program generates the proper response. When you activate the SQLQueryTableProducer component by calling its `Content` function, it initializes the SQLQuery component by obtaining the parameters from the HTTP request. The component can automatically examine the request method and then use either the `QueryFields` property (if the request is a `GET`) or the `ContentFields` property (if the request is a `POST`).

One problem with using a static HTML form as you did earlier is that it doesn't tell you which states and countries you can search for. To address this issue, you can use a selection control instead of an edit control in the HTML form. However, if the user adds a new record to the database table, you'll need to update the element list automatically. As a final solution, you can design the ISAPI DLL to produce a form on-the-fly, and you can fill the selection controls with the available elements.

You'll generate the HTML for this page in the `/form` action, which you connect to a PageProducer component. The PageProducer contains the following HTML text, which embeds two special tags:

```
<h4>Customer QueryProducer Search Form</h4>
<form action=" <#script>/search" method="POST">
<table>
<tr><td>State:</td>
  <td><select name="State"><option></option><#State_Province></select></td></tr>
<tr><td>Country:</td>
  <td><select name="Country"><option></option><#Country></select></td></tr>
```

```
<tr><td></td>
  <td><center><input type="Submit"></center></td></tr>
</table></form>
```

You'll notice that the tags have the same name as some of the table's fields. When the PageProducer encounters one of these tags, it adds an `<option>` HTML tag for every distinct value of the corresponding field. Here's the `OnTag` event handler's code, which is generic and reusable:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  ReplaceText := '';
  SQLQuery2.SQL.Clear;
  SQLQuery2.SQL.Add ('select distinct ' + TagString + ' from customer');
  try
    SQLQuery2.Open;
    try
      SQLQuery2.First;
      while not SQLQuery2.EOF do
      begin
        ReplaceText := ReplaceText +
          '<option>' + SQLQuery2.Fields[0].AsString + '</option>'#13;
        SQLQuery2.Next;
      end;
    finally
      SQLQuery2.Close;
    end;
  except
    ReplaceText := '{wrong field: ' + TagString + '}';
  end;
end;
```

This method uses a second SQLQuery component, which I manually placed on the form and connected to a shared SQLConnection component. It produces the output shown in Figure 20.4.

This web server extension, like many others, allows the user to view the details of a specific record. As in the previous example, you can accomplish this by customizing the output of the first column (column zero), which is generated by the QueryTableProducer component:

```
procedure TWebModule1.QueryTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + Request.ScriptName + '/record?Company=' + CellData +
      '">' + CellData + '</a>'#13;
  if CellData = '' then
    CellData := '&nbsp;';
end;
```

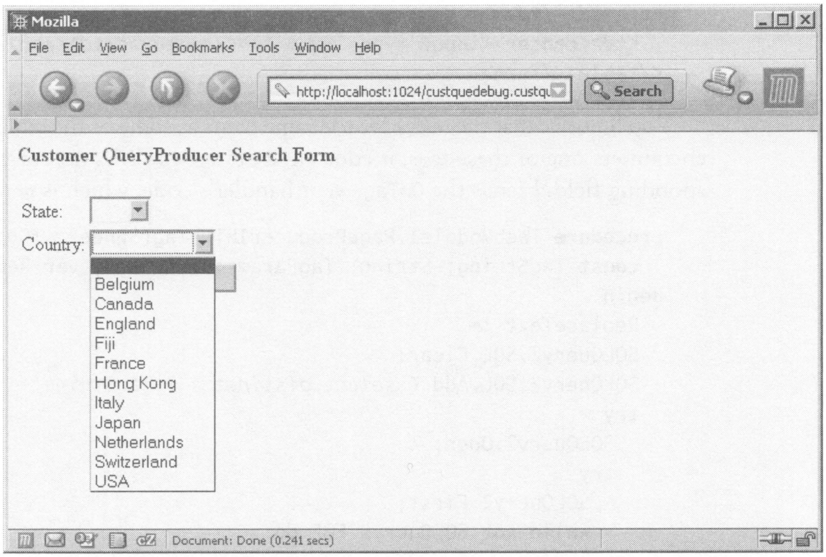

Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:1024/custquedebug.custqu
Search
Customer QueryProducer Search Form
State:
Country:
Belgium
Canada
England
Fiji
France
Hong Kong
Italy
Japan
Netherlands
Switzerland
USA
Document: Done (0.241 secs)

FIGURE 20.4
The form action of the CustQueP example produces an HTML form with a selection component dynamically updated to reflect the current status of the database.

TIP *When you have an empty cell in an HTML table, most browsers render it without the border. For this reason, I added a* non-breaking space *symbol (*`&nbsp;`*) in each empty cell. You'll have to do this in each HTML table generated with Delphi's table producers.*

The action for this link is `/record`, and you pass a specific element after the `?` parameter (without the parameter name, which is slightly nonstandard). The code you use to produce the HTML tables for the records doesn't use the producer components as you've been doing; instead, it shows the data of every field in a custom-built table:

```
procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  if Request.QueryFields.Count = 0 then
    Response.Content := 'Record not found'
  else
  begin
    Query2.SQL.Clear;
    Query2.SQL.Add ('select * from customer ' +
      'where Company="' + Request.QueryFields.Values['Company'] + '"');
    Query2.Open;
    Response.Content :=
      '<html><head><title>Customer Record</title></head><body>'#13 +
      '<h1>Customer Record: ' + Request.QueryFields[0] + '</h1>'#13 +
      '<table border>'#13;
    for I := 1 to Query2.FieldCount - 1 do
      Response.Content := Response.Content +
        '<tr><td>' + Query2.Fields [I].FieldName + '</td>'#13'<td>' +
        Query2.Fields [I].AsString + '</td></tr>'#13;
    Response.Content := Response.Content + '</table><hr>'#13 +
      // pointer to the query form
      '<a href="' + Request.ScriptName + '/form">' +
      ' Next Query </a>'#13 + '</body></html>'#13;
  end;
end;
```

Working with Apache

If you plan to use Apache instead of IIS or another web server, you can take advantage of the CGI technology to deploy your applications on almost any web server. However, using CGI means reduced speed and some trouble handling state information (because you cannot keep any data in memory). This is a good reason to write an ISAPI application or a dynamic Apache module. Using Delphi's WebBroker technology, you can also easily compile the same code for both technologies, so that moving your program to a different web platform becomes much simpler. You can also recompile a CGI program or a dynamic Apache module with Kylix and deploy it on a Linux server.

As I've mentioned, Apache can run traditional CGI applications but also has a specific technology for keeping the server extension program loaded in memory at all times for faster response. To build such a program in Delphi, you can use the Apache Shared Module option in the New Web Server Application dialog box; choose Apache 1 or Apache 2, depending on the version of the web server you plan to use.

WARNING *While D7 supports version 2.0.39 of Apache, it doesn't support the current popular version 2.0.40 due to a change in library interfaces. Use of version 2.0.39 is discouraged because it has a security problem. Information on how to fix the VCL code and make your modules compatible with Apache 2.0.40 and above have been posted by Borland R&D members on the newsgroups and are currently available also on Bob Swart's website at the URL* `www.drbob42.com/delphi7/apache2040.htm`.

If you choose to create an Apache module, you end up with a library having this type of source code for its project:

```
library Apache1;

uses
  WebBroker,
  ApacheApp,
  ApacheWm in 'ApacheWm.pas' {WebModule1: TWebModule};

{$R *.res}

exports
  apache_module name 'apache1_module';
```

```
begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Notice in particular the `exports` clause, which indicates the name used by Apache configuration files to reference the dynamic module. In the project source code, you can add two more definitions—the module name and the content type—in the following way:

```
ModuleName := 'Apache1_module';
ContentType:= 'Apache1-handler';
```

If you don't set these values, Delphi will assign them defaults, which are built adding the *_module* and *-handler* strings to the project name (resulting in the two names I used here).

An Apache module is generally not deployed within a script folder, but within the `modules` subfolder of the server (by default, `c:\Program Files\Apache\modules`). You have to edit the `http.conf` file, adding a line to load the module, as follows:

```
LoadModule apache1_module modules/apache1.dll
```

Finally, you must indicate when the module is invoked. The handler defined by the module can be associated with a given file extension (so that your module will process all the files having a given extension) or with a physical or virtual folder. In the latter case, the folder doesn't exist, but Apache pretends it is there. This is how you can set up a virtual folder for the Apache1 module:

```
<Location /Apache1>SetHandler Apache1-handler</Location>
```

Because Apache is inherently case sensitive (because of its Linux heritage), you also might want to add a second, lowercase virtual folder:

```
<Location /apache1>SetHandler Apache1-handler</Location>
```

Now you can invoke the sample application with the URL `http://localhost/Apache1`. A great advantage of using virtual folders in Apache is that a user doesn't really distinguish between the physical and dynamic portions of your site, as you can see by experimenting with the Apache1 example (which includes the code discussed here).

Practical Examples

After this general introduction to developing server-side applications with WebBroker, I'll end this part of the chapter with two practical examples. The first is a classic web counter. The second is an extension of the WebFind program presented in Chapter 19, which produces a dynamic page instead of filling a list box.

A Graphical Web Hit Counter

The server-side applications you've built up to now were based only on text. Of course, you can easily add references to existing graphics files. What's more interesting, however, is to build server-side programs capable of generating graphics that change over time.

A typical example is a *page hit counter*. To write a web counter, you save the current number of hits to a file and then read and increase the value every time the counter program is called. To return this information, all you need is HTML text with the number of hits. The code is straightforward:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  nHit: Integer;
  LogFile: Text;
  LogFileName: string;
begin
  LogFileName := 'WebCont.log';
  System.Assign (LogFile, LogFileName);
  try
    // read if the file exists
    if FileExists (LogFileName) then
    begin
      Reset (LogFile);
      Readln (LogFile, nHit);
      Inc (nHit);
    end
    else
      nHit := 0;
    // saves the new data
    Rewrite (LogFile);
    Writeln (LogFile, nHit);
  finally
    Close (LogFile);
  end;
  Response.Content := IntToStr (nHit);
end;
```

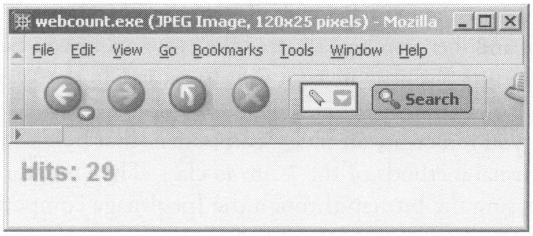
WARNING *This simple file handling does not scale. When multiple visitors hit the page at the same time, this code may return false results or fail with a file I/O error because a request in another thread has the file open for reading while this thread tries to open the file for writing. To support a similar scenario, you'll need to use a mutex (or a critical section in a multithreaded program) to let each subsequent thread wait until the thread currently using the file has completed its task.*

It's more interesting to create a graphical counter that can be easily embedded into any HTML page. There are two approaches to building a graphical counter: You can prepare a bitmap for each digit up front and then combine them in the program, or you can let the program draw over a memory bitmap to produce the graphic you want to return. In the WebCount program, I chose the second approach.

Basically, you can create an Image component that holds a memory bitmap, which you can paint on with the usual methods of the `TCanvas` class. Then you can attach this bitmap to a `TJpegImage` object. Accessing the bitmap through the JpegImage component converts the image to the JPEG format. Then, you can save the JPEG data to a stream and return it. As you can see, there are many steps, but the code is not complicated:

```
// create a bitmap in memory
Bitmap := TBitmap.Create;
try
  Bitmap.Width := 120;
  Bitmap.Height := 25;
  // draw the digits
  Bitmap.Canvas.Font.Name := 'Arial';
  Bitmap.Canvas.Font.Size := 14;
  Bitmap.Canvas.Font.Color := RGB (255, 127, 0);
  Bitmap.Canvas.Font.Style := [fsBold];
  Bitmap.Canvas.TextOut (1, 1, 'Hits: ' +
    FormatFloat ('###,###,###', Int (nHit)));
  // convert to JPEG and output
  Jpeg1 := TJpegImage.Create;
  try
    Jpeg1.CompressionQuality := 50;
    Jpeg1.Assign(Bitmap);
    Stream := TMemoryStream.Create;
    Jpeg1.SaveToStream (Stream);
    Stream.Position := 0;
    Response.ContentStream := Stream;
    Response.ContentType := 'image/jpeg';
    Response.SendResponse;
    // the response object will free the stream
  finally
    Jpeg1.Free;
  end;
finally
  Bitmap.Free;
end;
```

The three statements responsible for returning the JPEG image are the two that set the `ContentStream` and `ContentType` properties of the `Response` and the final call to `SendResponse`. The content type must match one of the possible MIME types accepted by the browser, and the order of these three statements is relevant. The `Response` object also has a `SendStream` method, but it should be called only after sending the type of the data with a separate call. Here you can see the effect of this program:



Hits: 29

To embed the program in a page, add the following code to the HTML:

```
<img src="http://localhost/scripts/webcount.exe" border=0 alt="hit counter">
```

Searching with a Web Search Engine

In Chapter 19, I discussed the use of the Indy HTTP client component to retrieve the result of a search on the Google website. Let's extend the example, turning it into a server-side application. The WebSearcher program, available as a CGI application or a Web App Debugger executable, has two actions: The first returns the HTML retrieved by the search engine; and the second parses the HTML filling a client data set component, which is hooked to a table page producer for generating the final output. Here is the code for the second action:

```
const
  strSearch = 'http://www.google.com/search?as_q=borland+delphi&num=100';

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  I: integer;
begin
  if not cds.Active then
    cds.CreateDataSet
  else
    cds.EmptyDataSet;
  for i := 0 to 5 do // how many pages?
  begin
    // get the data form the search site
    GrabHtml (strSearch + '&start=' + IntToStr (i*100));
    // scan it to fill the cds
    HtmlStringToCds;
  end;
  cds.First;
  // return producer content
  Response.Content := DataSetTableProducer1.Content;
end;
```

The `GrabHtml` method is identical to the WebFind example. The `HtlStringToCds` method is similar to the corresponding method of the WebFind example (which adds the items to a list box); it adds the addresses and their textual descriptions by calling

```
cds.InsertRecord ([0, strAddr, strText]);
```

The ClientDataSet component is set up with three fields: the two strings plus a line counter. This extra empty field is required in order to include the extra column in the table producer. The code fills the column in the cell-formatting event, which also adds the hyperlink:

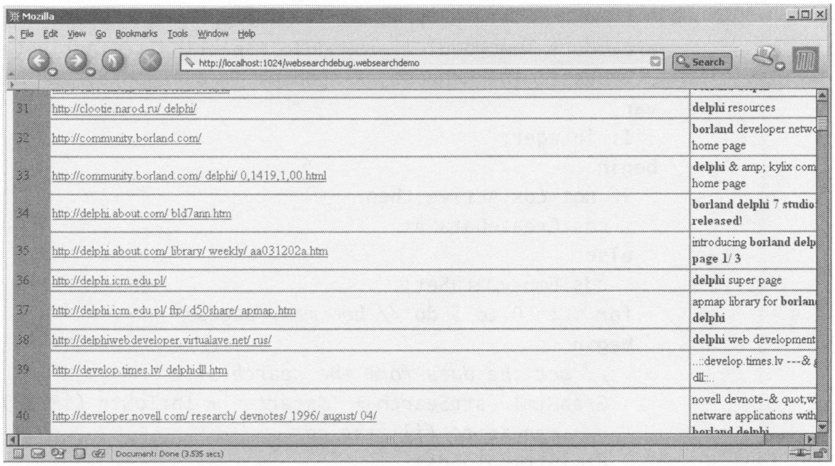
```
procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
```

```
    var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
  begin
    if CellRow <> 0 then
    case CellColumn of
      0: CellData := IntToStr (CellRow);
      1: CellData := '<a href="' + CellData + '">' + SplitLong(CellData) + '</a>';
      2: CellData := SplitLong (CellData);
    end;
  end;
```

The call to `SplitLong` is used to add extra spaces in the output text, to avoid having grid columns that are too large—the browser won't split the text on multiple lines unless it contains spaces or other special characters. The result of this program is a rather slow application (because of the multiple HTTP requests it must forward) that produces output like that shown in Figure 20.5.

FIGURE 20.5

The WebSearch program shows the result of multiple searches done on Google.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:1024/websearchdebug.websearchdemo
Search
Document: Done (3.535 secs)

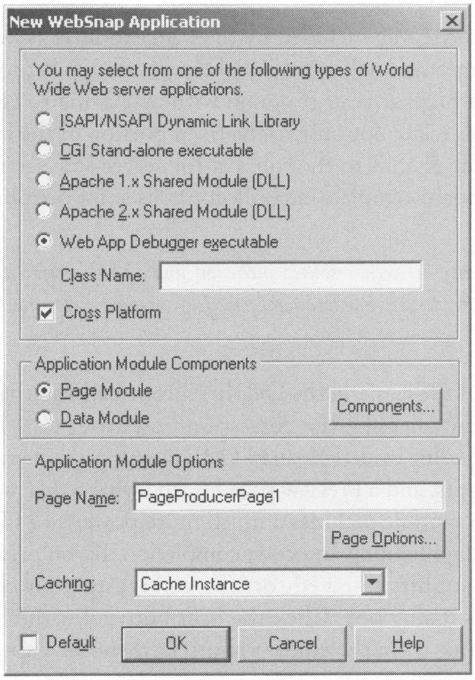
31	http://clootie.narod.ru/ delphi/	**delphi** resources
32	http://community.borland.com/	**borland** developer netwo home page
33	http://community.borland.com/ delphi/ 0,1419,1,00.html	**delphi** & amp; kylix com home page
34	http://delphi.about.com/ bld7ann.htm	**borland delphi 7 studio** **released!**
35	http://delphi.about.com/ library/ weekly/ aa031202a.htm	introducing **borland delp** **page 1/ 3**
36	http://delphi.icm.edu.pl/	**delphi** super page
37	http://delphi.icm.edu.pl/ ftp/ d50share/ apmap.htm	apmap library for **borlan** **delphi**
38	http://delphiwebdeveloper.virtualave.net/ rus/	**delphi** web development
39	http://develop.times.lv/ delphidll.htm	..::develop.times.lv ---& dll::..
40	http://developer.novell.com/ research/ devnotes/ 1996/ august/ 04/	novell devnote-& quot;w netware applications with **borland delphi**

WebSnap

Now that I've introduced the core elements of developing web server applications with Delphi, let's move to a more complex architecture available since Delphi 6: WebSnap. There are two good reasons I didn't jump into this topic at the beginning of this chapter. First, WebSnap builds on the foundation offered by WebBroker, so you cannot learn how to use the new features if you don't understand the underlying ones. For example, a WebSnap application is technically a CGI program, or an ISAPI or Apache module. Second, because WebSnap is included only in the Enterprise Studio version of Delphi, not all Delphi programmers have the chance to use it.

WebSnap has a few definite advantages over WebBroker, such as allowing for multiple web modules each corresponding to a page, integrating server-side scripting, XSL, and the Internet Express technology (these last two elements will be covered in Chapter 22, "Using XML Technologies"). Moreover, many ready-to-use components are available for handling common tasks, such as user logins, session management, and so on. Instead of listing all the features of WebSnap, though, I've decided to cover them in a sequence of simple and focused applications. I built these applications using the Web App Debugger, for testing purposes, but you can easily deploy them using one of the other available technologies.

When you're developing a WebSnap application, the starting point is a dialog box you can invoke either in the WebSnap page of the New Items dialog box (File ➢ New ➢ Other) or using the Internet toolbar in the IDE (which is not visible by default). The resulting dialog box, shown in Figure 20.6, allows you to choose the type of application (as in a WebBroker application) and to customize the initial application components (you'll be able to add more later). The bottom portion of the dialog determines the behavior of the first page (usually the default or home page of the program). A similar dialog box is displayed for subsequent pages.



New WebSnap Application

You may select from one of the following types of World Wide Web server applications.

○ ISAPI/NSAPI Dynamic Link Library
○ CGI Stand-alone executable
○ Apache 1.x Shared Module (DLL)
○ Apache 2.x Shared Module (DLL)
● Web App Debugger executable

Class Name:

☑ Cross Platform

Application Module Components

● Page Module
○ Data Module

Application Module Options

Page Name: PageProducerPage1

Caching: Cache Instance

☐ Default

FIGURE 20.6
The options offered by the New WebSnap Application dialog box include the type of server and a button that lets you select the core application components.

If you choose the defaults and type in a name for the home page, the dialog box will create a project and open a `TWebAppPageModule`. This module contains the components you've chosen, by default:

- A WebAppComponents component is a container for all the centralized services of the WebSnap application, such as the user list, core dispatcher, session services, and so on. Not all of its properties must be set up, because an application might not need all the available services.
- One of these core services is offered by the PageDispatcher component, which (automatically) holds a list of the application's available pages and defines the default page.
- Another core service is given by the AdapterDispatcher component, which handles HTML form submissions and image requests.

- The ApplicationAdapter is the first component you encounter from the *adapters* family. These components offer fields and actions to the server-side scripts evaluated by the program. Specifically, the ApplicationAdapter is a field adapter that exposes the value of its own `ApplicationTitle` property. If you enter a value for this property, it will be made available to the scripts.
- The module hosts a PageProducer that includes the HTML code of the page—in this case, the program's default page. Unlike in WebBroker applications, the HTML for this component is not stored inside its `HTMLDoc` string list property or referenced by its `HTMLFile` property. The HTML file is an external file, stored by default in the folder hosting the project's source code and referenced from the application using a statement similar to a resource include statement: `{*.html}`.
- Because the HTML file included by the PageProducer is kept as a separate file (the LocateFileService component will eventually help you for its deployment), you can edit it to change the output of a page of your program without having to recompile the application. These possible changes relate not only to the fixed portion of the HTML file but also to some of its dynamic content, thanks to the support for server-side scripting. The default HTML file, based on a standard template, already contains some scripting.

WARNING *The similarity between resource inclusion and HTML reference is mostly syntactical. The HTML reference is used only for the design-time location of the file, whereas a resource include directive links the data it refers into the executable file.*

You can view the HTML file in the Delphi editor thanks to that directive (with reasonably good syntax highlighting) by selecting the corresponding lower tab. The editor also has pages for a WebSnap module, including by default an HTML Result page where you can see the HTML generated after evaluating the scripts, and a Preview page hosting what a user will see in a browser. The Delphi 7 editor for a WebSnap module includes a much more powerful HTML editor than Delphi 6 had; it includes better syntax highlighting and code completion. If you prefer to edit your web application's HTML with a more sophisticated editor, you can set up your choice in the Internet page of the Environment Options dialog box. Click the Edit button for the HTML extension, and you can choose an external editor from the editor's shortcut menu or a specific button on Delphi's Internet toolbar.

The standard HTML template used by WebSnap adds to any page of the program its title and the application title, using script lines such as these:

```
<h1><%= Application.Title %></h1>
<h2><%= Page.Title %></h2>
```

We'll get back to the scripting in a while; we'll begin developing the WSnap1 example in the next section by creating a program with multiple pages. But first, I'll finish this overview by showing you the source code for a sample web page module:

```
type
  Thome = class(TWebAppPageModule)
    ...
  end;
```

```
function home: Thome;

implementation

{$R *.dfm}  {*.html}

uses WebReq, WebCntxt, WebFact, Variants;

function home: Thome;

begin
  Result := Thome(WebContext.FindModuleClass(Thome));
end;

initialization
  if WebRequestHandler <> nil then

    WebRequestHandler.AddWebModuleFactory(TWebAppPageModuleFactory.Create(
      Thome, TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html'),
        caCache));
end.
```

The module uses a global function instead of a form's typical global object to support page caching. This application also has extra code in the initialization section (particularly registration code) to let the application know the role of the page and its behavior.

TIP *Unlike this chapter's WebBroker examples, the WebSnap examples compile each in its own folder. This is because the HTML files are needed at runtime and I preferred simplifying the deployment as much as possible.*

Managing Multiple Pages

The first notable difference between WebSnap and WebBroker is that instead of having a single data module with multiple actions eventually connected to producer components, WebSnap has multiple data modules, each corresponding to an action and having a producer component with an HTML file attached to it. You can add multiple actions to a page/module, but the idea is that you structure applications around pages rather than actions. As is the case with actions, the name of the page is indicated in the request path.

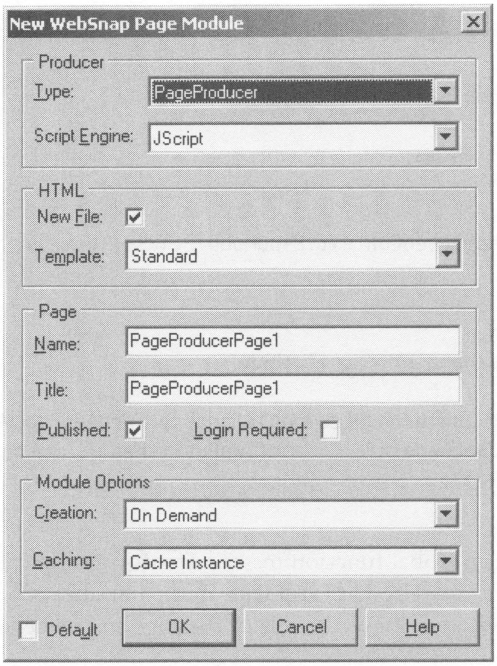
As an example, I added two more pages to the WebSnap application (which was built with default settings). For the first page, in the New WebSnap Page Module dialog (see Figure 20.7) choose a standard page producer and name it *date*. For the second, choose a DataSetPageProducer and name it *country*. After saving the files, you can begin testing the application. Thanks to some of the scripting I'll discuss later, each page lists all the available pages (unless you've unchecked the Published check box in the New WebSnap Page Module dialog).

The pages will be rather empty, but at least you have the structure in place. To complete the home page, you'll edit its linked HTML file directly. For the date page, use the same approach as a WebBroker application. Add some custom tags to the HTML text, like the following:

```
<p>The time at this site is <#time>.</p>
```

I also added code to the producer component's `OnTag` event handler to replace this tag with the current time.

FIGURE 20.7
The New WebSnap Page Module dialog box



Type: PageProducer
Script Engine: JScript
New File: ☑
Template: Standard
Name: PageProducerPage1
Title: PageProducerPage1
Published: ☑ Login Required: ☐
Creation: On Demand
Caching: Cache Instance
☐ Default

For the third page (the country page), modify the HTML to include tags for the various fields of the country table, as in:

```
<h3>Country: <#name></h3>
```

Then attach the ClientDataSet to the page producer:

```
object DataSetPageProducer: TDataSetPageProducer
  DataSet = cdsCountry
end
object cdsCountry: TClientDataSet
  FileName = 'C:\Program Files\Common Files\Borland
    Shared\Data\country.cds'
end
```

To open this dataset when the page is first created and reset it to the first record in further invocations, you handle the `OnBeforeDispatchPage` event of the web page module, adding this code to it:

```
cdsCountry.Open;
cdsCountry.First;
```

The fact that a WebSnap page can be very similar to a portion of a WebBroker application (basically, an action tied to a producer) is important, if you want to port existing WebBroker code to this new architecture. You can even port your existing DataSetTableProducer components to the new architecture. Technically, you can generate a new page, remove its producer component, replace it with a DataSetTableProducer, and hook this component to the `PageProducer` property of the web page module. In practice, this approach would cut out the HTML file of the page and its scripts.

In the WSnap1 program, I used a better technique. I added a custom tag (`<#htmltable>`) to the HTML file and used the `OnTag` event of the page producer to add to the HTML the result of the data set table:

```
if TagString = 'htmltable' then
  ReplaceText := DataSetTableProducer1.Content;
```

Server-Side Scripts

Having multiple pages in a server-side program—each associated with a different page module—changes the way you write a program. Having the server-side scripts at hand offers an even more powerful approach. For example, the standard scripts of the WSnap1 example account for the application and page titles, and for the index of the pages. This index is generated by an *enumerator*, the technique used to scan a list from within WebSnap script code. Let's look at it:

```
<table cellspacing="0" cellpadding="0"><td>
<%  e = new Enumerator(Pages)
    s = ''
    c = 0
    for (; !e.atEnd(); e.moveNext())
    {
      if (e.item().Published)
      {
        if (c > 0) s += '&nbsp;|&nbsp;'
        if (Page.Name != e.item().Name)
          s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
        else
          s += e.item().Title
        c++
      }
    }
    if (c>1) Response.Write(s)
%>
</td></table>
```

NOTE *Typically, WebSnap scripts are written in JavaScript, an object-based language common for Internet programming because it is the only scripting language generally available in browsers (on the client side). JavaScript (technically indicated as ECMAScript) borrows the core syntax of the C language and has almost nothing to do with Java. WebSnap uses Microsoft's ActiveScripting engine, which supports both JScript (a variation of JavaScript) and VBScript.*

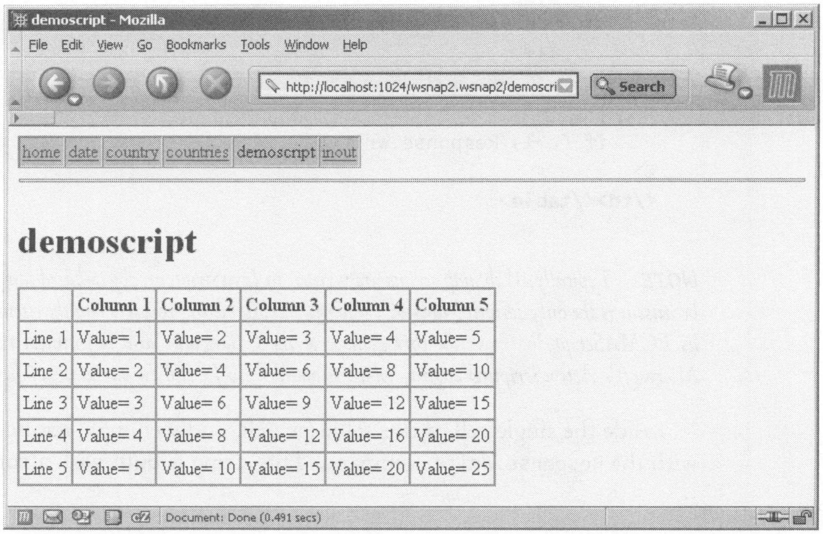
Inside the single cell of this table (which, oddly enough, has no rows), the script outputs a string with the `Response.Write` command. This string is built with a `for` loop over an enumerator of the application's pages, stored in the `Pages` global entity. The title of each page is added to the string only if the page is published. Each title uses a hyperlink with the exclusion of the current page. Having this code in a script, instead of hard-coded into a Delphi component, allows you to pass it to a good web designer, who can turn it into something a more visually appealing.

TIP *To publish or unpublish a page, don't look for a property in the web page module. This status is controlled by a flag of the* `AddWebModuleFactory` *method called in the web page module initialization code. You can comment or uncomment this flag to obtain the desired effect.*

As a sample of what you can do with scripting, I added to the WSnap2 example (an extension of the WSnap1 example) a *demoscript* page. The page's script can generate a full table of multiplied values with the following scripting code (see Figure 20.8 for its output):

```
<table border=1 cellspacing=0>
<tr>
  <th>&nbsp;</th>
  <% for (j=1;j<=5;j++) { %>
  <th>Column <%=j %></th>
  <% } %>
</tr>
<% for (i=1;i<=5;i++) { %>
<tr>
  <td>Line <%=i %></td>
  <% for (j=1;j<=5;j++) { %>
  <td>Value= <%=i*j %></td>
  <% } %>
</tr>
<% } %>
</table>
```

FIGURE 20.8
The WSnap2 example features a plain script and a custom menu stored in an include file.


demoscript - Mozilla
File
Edit
View
Go
Bookmarks
Tools
Window
Help
http://localhost:1024/wsnap2.wsnap2/demoscri
Search
home
date
country
countries
demoscript
inout
Document: Done (0.491 secs)

demoscript

	Column 1	Column 2	Column 3	Column 4	Column 5
Line 1	Value= 1	Value= 2	Value= 3	Value= 4	Value= 5
Line 2	Value= 2	Value= 4	Value= 6	Value= 8	Value= 10
Line 3	Value= 3	Value= 6	Value= 9	Value= 12	Value= 15
Line 4	Value= 4	Value= 8	Value= 12	Value= 16	Value= 20
Line 5	Value= 5	Value= 10	Value= 15	Value= 20	Value= 25

In this script, the `<%=` symbol replaces the longer `Response.Write` command. Another important feature of server-side scripting is the inclusion of pages within other pages. For example, if you plan to modify the menu, you can include the related HTML and script in a single file, instead of changing it and maintaining it in multiple pages. File inclusion is generally done with a statement like this:

```
<!-- #include file="menu.html" -->
```

In Listing 20.1, you can find the complete source code of the include file for the menu, which is referenced by all of the project's other HTML files. Figure 20.9 shows an example of this menu, which is displayed across the top of the page using the table-generation script mentioned earlier.

LISTING 20.1: THE `MENU.HTML` FILE INCLUDED IN EACH PAGE OF THE WSNAP2 EXAMPLE

```
<html>
<head>
<title><%= Page.Title %></title>
</head>
<body>
<h2><%= Application.Title %></h2>
<table cellspacing="0" cellpadding="2" border="1" bgcolor="#c0c0c0">
<tr>
<%  e = new Enumerator(Pages)
    for (; !e.atEnd(); e.moveNext())
    {
      if (e.item().Published)
      {
        if (Page.Name != e.item().Name)
          Response.Write ('<td><a href="' + e.item().HREF + '">' +
            e.item().Title + '</a></td>')
        else
          Response.Write ('<td>' + e.item().Title + '</td>')
      }
    }
%>
</tr>
</table>
<hr>
<h1><%= Page.Title %></h1>
<p>
```

This script for the menu uses the `Pages` list and the `Page` and `Application` global scripting objects. WebSnap makes available a few other global objects, including `EndUser` and `Session` objects (in case you add the corresponding adapters to the application), the `Modules` object, and the `Producer` object, which allows access to the Producer component of the web page module. The script also has available the `Response` and `Request` objects of the web module.

Adapters

In addition to these global objects, within a script you can access all the adapters available in the corresponding web page module. (Adapters in other modules, including shared web data modules, must be referenced by prefixing their name with the `Modules` object and the corresponding module.) Adapters allow you to pass information from your compiled Delphi code to the interpreted script, providing a scriptable interface to your Delphi application. Adapters contain fields that represent data and host actions that represent commands. The server-side scripts can access these values and issue these commands, passing specific parameters to them.

ADAPTER FIELDS

For simple customizations, you can add new fields to specific adapters. For instance, in the WSnap2 example, I added a custom field to the application adapter. After selecting this component, you can either open its Fields editor (accessible via its shortcut menu) or work in the Object TreeView. After adding a new field (called `AppHitCount` in the example), you can assign a value to it in its `OnGetValue` event. Because you want to count the hits (or requests) on any page of the web application, you can also handle the `OnBeforePageDispatch` event of the *global* PageDispatcher component to increase the value of a local field, `HitCount`. Here is the code for the two methods:

```
procedure Thome.PageDispatcherBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
begin
  Inc (HitCount);
end;

procedure Thome.AppHitCountGetValue(Sender: TObject; var Value: Variant);
begin
  Value := HitCount;
end;
```

Of course, you could use the page name to also count hits on each page (and you could add support for persistency, because the count is reset every time you run a new instance of the application). Now that you've added a custom field to an existing adapter (corresponding to the `Application` script object), you can access it from within any script, like this:

```
<p>Application hits since last activation:
<%= Application.AppHitCount.Value %></p>
```

ADAPTER COMPONENTS

In the same way, you can add custom adapters to specific pages. If you need to pass along a few fields, use the generic Adapter component. Other custom adapters (besides the global ApplicationAdapter you've already used) include these:

- The PagedAdapter component has built-in support for showing its content over multiple pages.
- The DataSetAdapter component is used to access a Delphi dataset from a script and is covered in the section "WebSnap and Databases."

- The StringValuesList holds a list of name/value pairs, such as a string list, and can be used directly or to provide a list of values to an adapter field. The inherited DataSetValuesList adapter has the same role but grabs the list of name/value pairs from a dataset, providing support for lookups and other selections.
- User-related adapters, such as the EndUser, EndUserSession, and LoginForm adapters, are used to access user and session information and to build a login form for the application that is automatically tied to the users list. I'll discuss these adapters in the section "Sessions, Users, and Permissions" later in this chapter.

USING THE ADAPTERPAGEPRODUCER

Most of these components are used in conjunction with an AdapterPageProducer component. The AdapterPageProducer can generate portions of script after you visually design the desired result. As an example, I've added to the WSnap2 application the *inout* page, which has an adapter with two fields, one standard and one Boolean:

```
object Adapter1: TAdapter
  OnBeforeExecuteAction = Adapter1BeforeExecuteAction
  object TAdapterActions
    object AddPlus: TAdapterAction
      OnExecute = AddPlusExecute
    end
    object Post: TAdapterAction
      OnExecute = PostExecute
    end
  end
  object TAdapterFields
    object Text: TAdapterField
      OnGetValue = TextGetValue
    end
    object Auto: TAdapterBooleanField
      OnGetValue = AutoGetValue
    end
  end
end
```

The adapter also has a couple of actions that post the current user input and add a plus sign (+) to the text. The same plus sign is added when the `Auto` field is enabled. Developing the user interface for this form and the related scripting would take some time using plain HTML. But the AdapterPageProducer component (used in this page) has an integrated HTML designer, which Borland calls Web Surface Designer. Using this tool, you can visually add a form to the HTML page and add an AdapterFieldGroup to it. Connect this field group to the adapter to automatically display editors for the two fields. Then you can add an AdapterCommandGroup and connect it to the AdapterFieldGroup, to provide buttons for all of the adapter's actions. You can see an example of this designer in Figure 20.9.

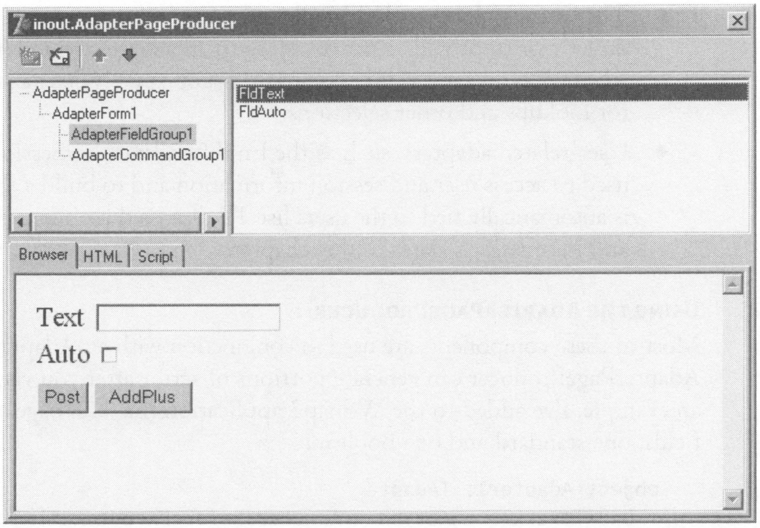

inout.AdapterPageProducer
AdapterPageProducer
AdapterForm1
AdapterFieldGroup1
AdapterCommandGroup1
FldText
FldAuto
Browser
HTML
Script
Text
Auto
Post
AddPlus

FIGURE 20.9 The Web Surface Designer for the inout page of the WSnap2 example, at design time

To be more precise, the fields and buttons are automatically displayed if the `AddDefaultFields` and `AddDefaultCommands` properties of the field group and command group are set. The effect of these visual operations to build this form are summarized in the following DFM snippet:

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = Adapter1
      object FldText: TAdapterDisplayField
        FieldName = 'Text'
      end
      object FldAuto: TAdapterDisplayField
        FieldName = 'Auto'
      end
    end
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup1
      object CmdPost: TAdapterActionButton
        ActionName = 'Post'
      end
      object CmdAddPlus: TAdapterActionButton
        ActionName = 'AddPlus'
      end
    end
  end
end
```

Now that you have an HTML page with some scripts to move data back and forth and issue commands, let's look at the source code required to make this example work. First, you must add to the class two local fields to store the adapter fields and manipulate them, and you need to implement the `OnGetValue` event for both, returning the field values. When each button is clicked, you must retrieve the text passed by the user, which is not automatically copied into the corresponding adapter field. You can obtain this effect by looking at the `ActionValue` property of these fields, which is set only if something was entered (for this reason, when nothing is entered you set the Boolean field to False). To avoid repeating this code for both actions, place it in the `OnBeforeExecuteAction` event of the web page module:

```
procedure Tinout.Adapter1BeforeExecuteAction(Sender, Action: TObject;
  Params: TStrings; var Handled: Boolean);
begin
  if Assigned (Text.ActionValue) then
    fText := Text.ActionValue.Values [0];
  fAuto := Assigned (Auto.ActionValue);
end;
```

Notice that each action can have multiple values (in case components allow multiple selections); but this is not the case, so you can grab the first element. Finally, here is the code for the `OnExecute` events of the two actions:

```
procedure Tinout.AddPlusExecute(Sender: TObject; Params: TStrings);
begin
  fText := fText + '+';
end;

procedure Tinout.PostExecute(Sender: TObject; Params: TStrings);
begin
  if fAuto then
    AddPlusExecute (Self, nil);
end;
```

As an alternative, adapter fields have a public `EchoActionFieldValue` property you can set to get the value entered by the user and place it in the resulting form. This technique is typically used in case of errors, to let the user change the input starting with the values already entered.

NOTE *The AdapterPageProducer component has specific support for Cascading Style Sheets (CSS). You can define the CSS for a page using either the* `StylesFile` *property or the* `Styles` *string list. Any element of the editor of the producer's items can define a specific style or choose a style from the attached CSS. You accomplish this last operation (which is the suggested approach) using the* `StyleRule` *property.*

SCRIPTS RATHER THAN CODE?

Even this example of the combined use of an adapter and an adapter page producer, with its visual designer, shows the power of this architecture. However, this approach also has a drawback: By letting the components generate the script (in the HTML, you have only the `<#SERVERSCRIPT>` tag), you save a lot of development time; but you end up mixing the script with the code, so changes to the user interface will require updating the program. The division of responsibilities between the Delphi application developer and the HTML/script designer is lost. And, ironically, you end up having to run a script to accomplish something the Delphi program could have done right away, possibly much faster.

In my opinion, WebSnap is a powerful architecture and a huge step forward from WebBroker, but it must be used with care to avoid misusing some of these technologies just because they are simple and powerful. For example, it might be worth using the AdapterPageProducer designer to generate the first version of a page, and then grabbing the generated script and copying it to a plain PageProducer's HTML, so that a web designer can modify the script with a specific tool.

For nontrivial applications, I prefer the possibilities offered by XML and XSL, which are available within this architecture even if they don't have a central role. You'll find more on this topic in Chapter 22.

Locating Files

When you have written an application like the one just described, you must deploy it as a CGI or dynamic library. Instead of placing the templates and include files in the same folder as the executable file, you can devote a subfolder or custom folder to host all the files. The LocateFileService component handles this task.

The component is not intuitive to use. Instead of having you specify a target folder as a property, the system fires one of this component's events any time it has to locate a file. (This approach is much more powerful.) There are three events: `OnFindIncludeFile`, `OnFindStream`, and `OnFindTemplateFile`. The first and last events return the name of the file to use in a `var` parameter. The `OnFindStream` event allows you to provide a stream directly, using one you already have in memory or that you've created on the fly, extracted from a database, obtained via an HTTP connection, or gotten any other way you can think of. In the simplest case of the `OnFindIncludeFile` event, you can write code like the following:

```
procedure TPageProducerPage2.LocateFileService1FindIncludeFile(
  ASender: TObject; AComponent: TComponent; const AFileName: String;
  var AFoundFile: String; var AHandled: Boolean);
begin
  AFoundFile := DefaultFolder + AFileName;
  AHandled := True;
end;
```

WebSnap and Databases

Delphi has always shone in the area of database programming. For this reason, it is not surprising to see a lot of support for handling datasets within the WebSnap framework. Specifically, you can use the DataSetAdapter component to connect to a dataset and display its values in a form or a table using the AdapterPageProducer component's visual editor.

A WebSnap Data Module

As an example, I built a new WebSnap application (called WSnapTable) with an AdapterPageProducer as its main page to display a table in a grid and another AdapterPageProducer in a secondary page to show a form with a single record. I also added to the application a WebSnap Data Module, as a container for the dataset components. The data module has a ClientDataSet that's wired to a dbExpress dataset through a provider and based on an InterBase connection, as shown here:

```
object ClientDataSet1: TClientDataSet
  Active = True
  ProviderName = 'DataSetProvider1'
end
object SQLConnection1: TSQLConnection
  Connected = True
  ConnectionName = 'IBLocal'
  LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
  SQLConnection = SQLConnection1
  CommandText =
    'select CUST_NO, CUSTOMER, ADDRESS_LINE1, CITY, STATE_PROVINCE, ' +
    '  COUNTRY from CUSTOMER'
end
object DataSetProvider1: TDataSetProvider
  DataSet = SQLDataSet1
end
```

The DataSetAdapter

Now that you have a dataset available, you can add a DataSetAdapter to the first page and connect it to the web module's ClientDataSet. The adapter automatically makes available all of the dataset's fields and several predefined actions for operating on the dataset (such as Delete, Edit, and Apply). You can add them explicitly to the `Actions` and `Fields` collections to exclude some of them and customize their behavior, but this step is not always required.

Like the PagedAdapter, the DataSetAdapter has a `PageSize` property you can use to indicate the number of elements to display in each page. The component also has commands you can use to navigate among pages. This approach is particularly suitable when you want to display a large dataset in a grid. These are the adapter settings for the main page of the WSnapTable example:

```
object DataSetAdapter1: TDataSetAdapter
  DataSet = WebDataModule1.ClientDataSet1
  PageSize = 6
end
```

The corresponding page producer has a form containing two command groups and a grid. The first command group (displayed above the grid) has the predefined commands for handling pages: `CmdPrevPage`, `CmdNextPage`, and `CmdGotoPage`. The last command generates a list of numbers for the pages, so that a user can jump to each of them directly. The AdapterGrid component has the default columns plus an extra column hosting Edit and Delete commands. The bottom command group provides a button used to create a new record. You can see an example of the table's output in Figure 20.10 and the complete settings of the AdapterPageProducer in Listing 20.2.

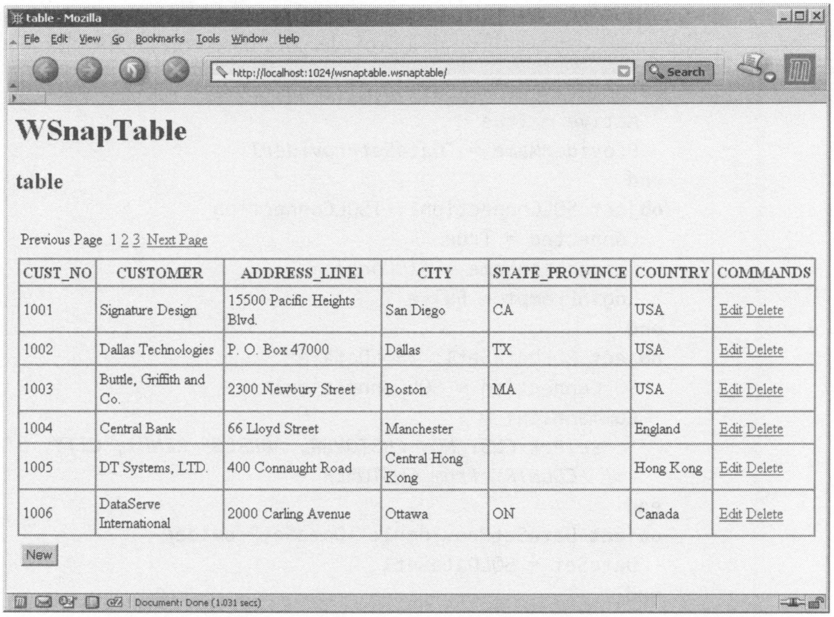


table - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:1024/wsnaptable.wsnaptable/
Search
New
Document: Done (1.031 secs)

FIGURE 20.10 The page shown by the WSnapTable example at startup includes the initial portion of a paged table.

WSnapTable

table

Previous Page 1 2 3 Next Page

CUST_NO	CUSTOMER	ADDRESS_LINE1	CITY	STATE_PROVINCE	COUNTRY	COMMANDS
1001	Signature Design	15500 Pacific Heights Blvd.	San Diego	CA	USA	Edit Delete
1002	Dallas Technologies	P. O. Box 47000	Dallas	TX	USA	Edit Delete
1003	Buttle, Griffith and Co.	2300 Newbury Street	Boston	MA	USA	Edit Delete
1004	Central Bank	66 Lloyd Street	Manchester		England	Edit Delete
1005	DT Systems, LTD.	400 Connaught Road	Central Hong Kong		Hong Kong	Edit Delete
1006	DataServe International	2000 Carling Avenue	Ottawa	ON	Canada	Edit Delete

LISTING 20.2: ADAPTERPAGEPRODUCER SETTINGS FOR THE WSNAPTABLE MAIN PAGE

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterGrid1
      object CmdPrevPage: TAdapterActionButton
        ActionName = 'PrevPage'
        Caption = 'Previous Page'
      end
      object CmdGotoPage: TAdapterActionButton...
      object CmdNextPage: TAdapterActionButton
        ActionName = 'NextPage'
        Caption = 'Next Page'
      end
    end
    object AdapterGrid1: TAdapterGrid
      TableAttributes.CellSpacing = 0
      TableAttributes.CellPadding = 3
      Adapter = DataSetAdapter1
      AdapterMode = 'Browse'
      object ColCUST_NO: TAdapterDisplayColumn
      ...
```

```
          object AdapterCommandColumn1: TAdapterCommandColumn
            Caption = 'COMMANDS'
            object CmdEditRow: TAdapterActionButton
              ActionName = 'EditRow'
              Caption = 'Edit'
              PageName = 'formview'
              DisplayType = ctAnchor
            end
            object CmdDeleteRow: TAdapterActionButton
              ActionName = 'DeleteRow'
              Caption = 'Delete'
              DisplayType = ctAnchor
            end
          end
        end
        object AdapterCommandGroup2: TAdapterCommandGroup
          DisplayComponent = AdapterGrid1
          object CmdNewRow: TAdapterActionButton
            ActionName = 'NewRow'
            Caption = 'New'
            PageName = 'formview'
          end
        end
      end
    end
```

There are a couple of things to notice in this listing. First, the grid's `AdapterMode` property is set to Browse (other possibilities are Edit, Insert, and Query). This dataset display mode for adapters determines the type of user interface (text or edit boxes and other input controls) and the visibility of other buttons (for example, Apply and Cancel buttons are only present in the edit view; the opposite is true for the Edit command).

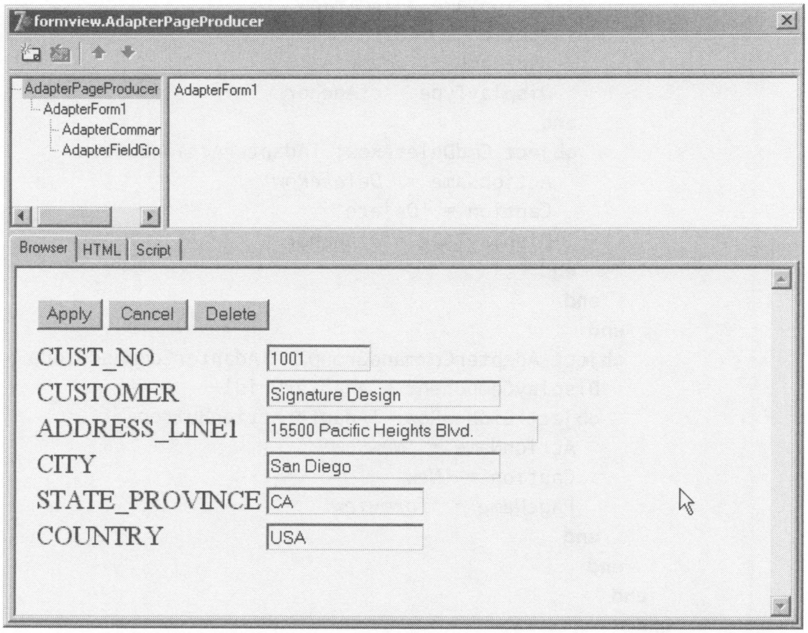
NOTE *You can also modify the adapter mode by using server-side script and accessing* `Adapter.Mode`.

Second, I modified the display of the commands in the grid using the ctAnchor value for the `DisplayType` property instead of the default button style. Similar properties are available for most components of this architecture to tweak the HTML code they produce.

Editing the Data in a Form

Some of the commands are connected to a different page, which will be displayed after the commands are invoked. For example, the `edit` command's `PageName` property is set to formview. This second page of the application has an AdapterPageProducer with components hooked to the same DataSetAdapter as of the other table, so that all the requests will be automatically synchronized. If you click the Edit command, the program will open the secondary page that displays the data of the record corresponding to the command.

Listing 20.3 shows the details of the page producer for the program's second page. Again, building the HTML form visually using the Delphi-specific designer (see Figure 20.11) was a very fast operation.


formview.AdapterPageProducer
AdapterPageProducer
AdapterForm1
AdapterCommar
AdapterFieldGro
AdapterForm1
Browser
HTML
Script
Apply
Cancel
Delete

CUST_NO 1001
CUSTOMER Signature Design
ADDRESS_LINE1 15500 Pacific Heights Blvd.
CITY San Diego
STATE_PROVINCE CA
COUNTRY USA

FIGURE 20.11 The formview page shown by the WSnapTable example at design time, in the Web Surface Designer (or AdapterPageProducer editor)

LISTING 20.3: ADAPTERPAGEPRODUCER SETTINGS FOR THE FORMVIEW PAGE

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterErrorList1: TAdapterErrorList
      Adapter = table.DataSetAdapter1
    end
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup1
      object CmdApply: TAdapterActionButton
        ActionName = 'Apply'
        PageName = 'table'
      end
      object CmdCancel: TAdapterActionButton
        ActionName = 'Cancel'
        PageName = 'table'
      end
      object CmdDeleteRow: TAdapterActionButton
        ActionName = 'DeleteRow'
        Caption = 'Delete'
        PageName = 'table'
      end
    end
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = table.DataSetAdapter1
      AdapterMode = 'Edit'
      object FldCUST_NO: TAdapterDisplayField
        DisplayWidth = 10
        FieldName = 'CUST_NO'
      end
      object FldCUSTOMER: TAdapterDisplayField
        ...
    end
  end
end
```

In the listing, you can see that all the operations send the user back to the main page and that the `AdapterMode` is set to Edit, unless there are update errors or conflicts. In this case, the same page is displayed again, with a description of the errors obtained by adding an AdapterErrorList component at the top of the form.

The second page is not published, because selecting it without referring to a specific record would make little sense. To unpublish the page, you comment the corresponding flag in the initialization code. Finally, to make the changes to the database persistent, you can call the `ApplyUdpates` method in the `OnAfterPost` and `OnAfterDelete` events of the ClientDataSet component hosted by the data module.

Another problem (which I haven't fixed) relates to the fact that the SQL server assigns the ID of each customer, so that when you enter a new record, the data in the ClientDataSet and in the database are no longer aligned. This situation can cause Record Not Found errors.

Master/Detail in WebSnap

The DataSetAdapter component has specific support for master/detail relationships between datasets. After you've created the relationship among the datasets, as usual, define an adapter for each dataset and then connect the `MasterAdapter` property of the detail dataset's adapter. Setting up the master/detail relationship between the adapters makes them work in a more seamless way. For example, when you change the work mode of the master or enter new records, the detail automatically enters into Edit mode or is refreshed.

The WSnapMD example uses a data module to define such a relationship. It includes two ClientDataSet components, each connected to a SQLDataSet through a provider. The data access components each refer to a table, and the ClientDataSet components define a master/detail relationship. The same data module hosts two dataset adapters that refer to the two datasets and again define the master/detail relationship:

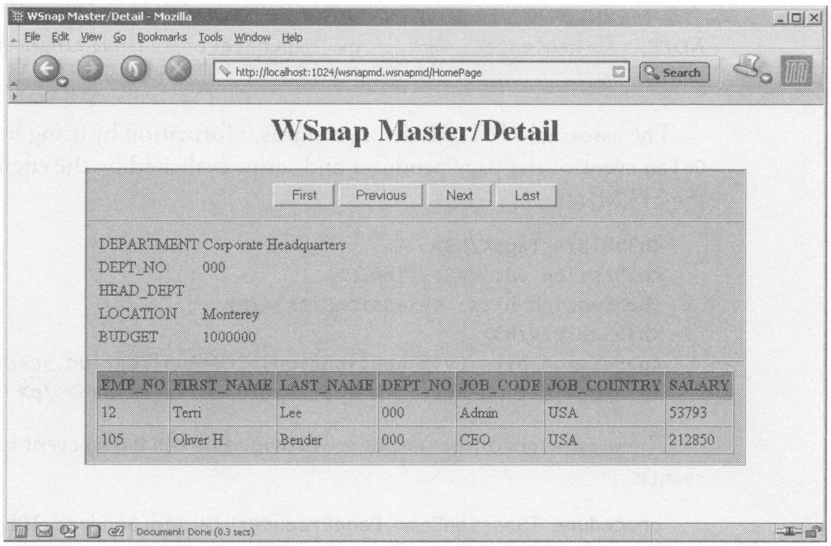
```
object dsaDepartment: TDataSetAdapter
  DataSet = cdsDepartment
end
```

```
object dsaEmployee: TDataSetAdapter
  DataSet = cdsEmployee
  MasterAdapter = dsaDepartment
end
```

WARNING *I originally tried to use a SimpleDataSet component to avoid cluttering the data module, but this approach didn't work. The master/detail portion of the program was correct, but moving from a page to the next or a previous page with the related buttons kept failing. The reason is that if you use a SimpleDataSet, a bug closes the dataset at each interaction, losing the status information.*

This WebSnap application's only page has an AdapterPageProducer component hooked to both dataset adapters. The page's form has a field group hooked to the master and a grid connected with the detail. Unlike other examples, I tried to improve the user interface by adding custom attributes for the various elements. I used a gray background, displayed some of the grid borders (HTML grids are often used by Web Surface Designer), centered most of the elements, and added spacing. Notice that I added extra spaces to the button captions to prevent them from being too small. You can see the related code in the following detailed excerpt and the effect in Figure 20.12:

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    Custom = 'Border="1" CellSpacing="0" CellPadding="10" ' +
      'BgColor="Silver" align="center"'
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup1
      Custom = 'Align="Center"'
      object CmdFirstRow: TAdapterActionButton...
      object CmdPrevRow: TAdapterActionButton...
      object CmdNextRow: TAdapterActionButton...
      object CmdLastRow: TAdapterActionButton...
    end
    object AdapterFieldGroup1: TAdapterFieldGroup
      Custom = 'BgColor="Silver"'
      Adapter = WDataMod.dsaDepartment
      AdapterMode = 'Browse'
    end
    object AdapterGrid1: TAdapterGrid
      TableAttributes.BgColor = 'Silver'
      TableAttributes.CellSpacing = 0
      TableAttributes.CellPadding = 3
      HeadingAttributes.BgColor = 'Gray'
      Adapter = WDataMod.dsaEmployee
      AdapterMode = 'Browse'
      object ColEMP_NO: TAdapterDisplayColumn...
      object ColFIRST_NAME: TAdapterDisplayColumn...
      object ColLAST_NAME: TAdapterDisplayColumn...
      object ColDEPT_NO: TAdapterDisplayColumn...
      object ColJOB_CODE: TAdapterDisplayColumn...
      object ColJOB_COUNTRY: TAdapterDisplayColumn...
      object ColSALARY: TAdapterDisplayColumn...
    end
  end
end
```


WSnap Master/Detail - Mozilla
File
Edit
View
Go
Bookmarks
Tools
Window
Help
http://localhost:1024/wsnapmd.wsnapmd/HomePage
Search
Document: Done (0.3 secs)

WSnap Master/Detail

First | Previous | Next | Last

DEPARTMENT Corporate Headquarters
DEPT_NO 000
HEAD_DEPT
LOCATION Monterey
BUDGET 1000000

EMP_NO	FIRST_NAME	LAST_NAME	DEPT_NO	JOB_CODE	JOB_COUNTRY	SALARY
12	Terri	Lee	000	Admin	USA	53793
105	Oliver H.	Bender	000	CEO	USA	212850

FIGURE 20.12
The WSnapMD example shows a master/detail structure and has some customized output.

Sessions, Users, and Permissions

Another interesting feature of the WebSnap architecture is its support for sessions and users. Sessions are supported using a classic approach: temporary cookies. These cookies are sent to the browser so that following requests from the same user can be acknowledged by the system. By adding data to a session instead of an application adapter, you can have data that depends on the specific session or user (although a user can run multiple sessions by opening multiple browser windows on the same computer). For supporting sessions, the application keeps data in memory, so this feature is not available in CGI programs.

Using Sessions

To underline the importance of this type of support, I built a WebSnap application with a single page showing both the total number of hits and the total number of hits for each session. The program has a SessionService component with default values for its `MaxSessions` and `DefaultTimeout` properties. For every new request, the program increases both an `nHits` private field of the page module and the `SessionHits` value of the current session:

```
procedure TSessionDemo.WebAppPageModuleBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
```

```
begin
  // increase application and session hits
  Inc (nHits);
  WebContext.Session.Values ['SessionHits'] :=
    Integer (WebContext.Session.Values ['SessionHits']) + 1;
end;
```

NOTE *The* `WebContext` *object (of type* `TWebContext`*) is a thread variable created by WebSnap for each request. It provides thread-safe access to other global variables used by the program.*

The associated HTML displays status information by using both custom tags evaluated by the `OnTag` event of the page producer and script evaluated by the engine. Here is the core portion of the HTML file:

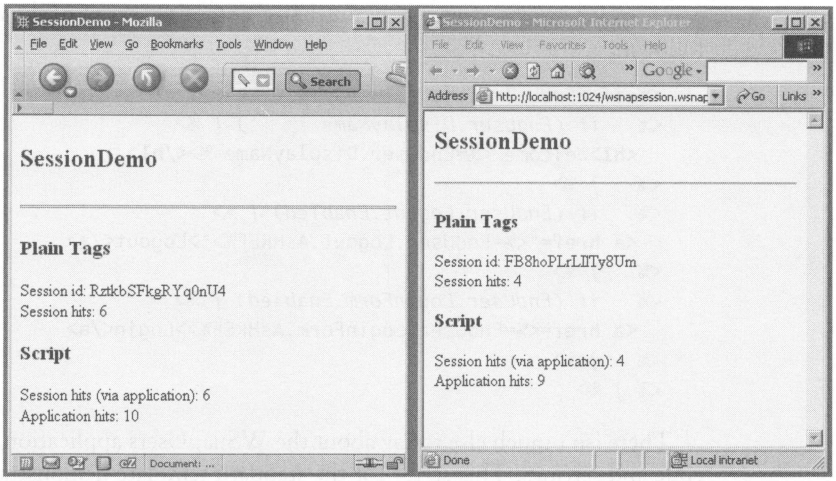
```
<h3>Plain Tags</h3>
<p>Session id: <#SessionID>
<br>Session hits: <#SessionHits></p>
<h3>Script</h3>
<p>Session hits (via application): <%=Application.SessionHits.Value%>
<br>Application hits: <%=Application.Hits.Value%></p>
```

The parameters of the output are provided by the `OnTag` event handler and the fields' `OnGetValue` events:

```
procedure TSessionDemo.PageProducerHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'SessionID' then
    ReplaceText := WebContext.Session.SessionID
  else if TagString = 'SessionHits' then
    ReplaceText := WebContext.Session.Values ['SessionHits']
end;

procedure TSessionDemo.HitsGetValue(Sender: TObject; var Value: Variant);
begin
  Value := nHits;
end;

procedure TSessionDemo.SessionHitsGetValue(Sender: TObject; var Value: Variant);
begin
  Value := Integer (WebContext.Session.Values ['SessionHits']);
end;
```

The effect of this program is visible in Figure 20.13, where I've activated two sessions in two different browsers.

TIP *In this example, I used both the traditional WebBroker tag replacement and the newer WebSnap adapter fields and scripting, so that you can compare the two approaches. Keep in mind that they are both available in a WebSnap application.*


SessionDemo - Mozilla
File Edit View Go Bookmarks Tools Window Help
Search
Document: ...
SessionDemo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Google
Address http://localhost:1024/wsnapsession.wsnap
Go
Links
Done
Local intranet

SessionDemo

Plain Tags

Session id: RztkbSFkgRYq0nU4
Session hits: 6

Script

Session hits (via application): 6
Application hits: 10

SessionDemo

Plain Tags

Session id: FB8hoPLrLlITy8Um
Session hits: 4

Script

Session hits (via application): 4
Application hits: 9

FIGURE 20.13
Two instances of the browser operate on two different sessions of the same WebSnap application.

Requesting Login

In addition to generic sessions, WebSnap also has specific support for users and login-based authorized sessions. You can add to an application a list of users (with the WebUserList component), each with a name and a password. This component is rudimentary in the data it can store. However, instead of filling it with your list of users, you can keep the list in a database table (or in another proprietary format) and use the WebUserList component's events to retrieve your custom user data and check the user passwords.

You'll generally also add to the application the SessionService and EndUserSessionAdapter components. At this point, you can ask the users to log in, indicating for each page whether it can be viewed by everyone or only by logged-in users. This is accomplished by setting the wpLoginRequired flag in the constructor of the `TWebPageModuleFactory` and `TWebAppPageModuleFactory` classes in the web page unit's initialization code.

NOTE *Rights and publication information is included in the factory rather than in the WebPageModule because the program can check the access rights and list the pages even without loading the module.*

When a user tries to see a page that requires user identification, the login page indicated in the EndUserSessionAdapter component is displayed. You can create such a page easily by creating a new web page module based on an AdapterPageProducer and adding to it the LoginFormAdapter. In the page's editor, add a field group within a form, connect the field group to the LoginFormAdapter, and add a command group with the default Login button. The resulting login form will have fields for the username and its password, and also for the requested page. This last value is automatically filled with the requested page, in case this page required authorization and the user wasn't already logged in. This way, a user can immediately reach the requested page without being bounced back to a generic menu.

The login form is typically not published, because the corresponding Login command is already available when the user isn't logged in to the system; when the user logs in, it is replaced by a Logout command. This command is obtained by the standard script of the web page module, particularly the following:

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
<%   if (EndUser.Logout.Enabled) { %>
  <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
  <a href=<%=EndUser.LoginForm.AsHREF%>>Login</a>
<%   } %>
<% } %>
```

There isn't much else to say about the WSnapUsers application, because it has almost no custom code and settings. This script for the standard template demonstrates the access to the user data.

SINGLE PAGE ACCESS RIGHTS

In addition to making pages require a login for access, you can give specific users the right to see more pages than others. Any user has a set of rights separated by semicolons or commas. The user must have all the rights defined for the requested page. These rights, which are generally listed in the `ViewAccess` and `ModifyAccess` properties of the adapters, indicate respectively whether the user can see the given elements while browsing or can edit them. These settings are granular and can be applied to entire adapters or specific adapter fields (notice I'm referring to the adapter fields, not the user interface components within the designer). For example, you can hide some of a table's columns from given users by hiding the corresponding fields (and also in other cases, as specified by the `HideOptions` property).

The global PageDispatcher component also has `OnCanViewPage` and `OnPageAccessDenied` events, which you can use to control access to various pages of a program within the program code, allowing for even greater control.

What's Next?

In this chapter, I've covered web server applications, using multiple server-side techniques and two different frameworks of the Delphi class library: WebBroker and WebSnap. This wasn't an in-depth presentation, because I could write an entire book on this topic alone. It was intended as a starting point, and (as usual) I've tried to make the core concepts clear rather than building complex examples.

If you want to learn more about the WebSnap framework and see examples in actions, refer to the extensive Delphi demos in the `\Demos\WebSnap` folder. Some other options, relating to XML, XSL, and client-side scripts, will be examined in the Chapter 22.

Chapter 21 is devoted to another alternative framework for developing web server applications in Delphi: IntraWeb. This third-party tool has been around for a few years, but it has now become more relevant because Borland included it in Delphi 7. As you'll see, IntraWeb can be integrated in a WebBroker or WebSnap program, or it can be used as a separate architecture.

Chapter 21

Web Programming with IntraWeb

SINCE DELPHI 2 DAYS, Chad Z. Hower has been building a Delphi architecture for simplifying the development of web applications, with the idea of making web programming as simple and as visual as standard Delphi form programming. Some programmers are fully acquainted with dynamic HTML, JavaScript, Cascading Style Sheets, and the latest Internet technologies. Other programmers just want to build web applications in Delphi the way they build VCL or CLX applications.

IntraWeb is intended for this second category of developers, although it is so powerful that even expert web programmers can benefit from its use. In Chad's words, IntraWeb is for building web applications, not websites. Moreover, IntraWeb components can be used in a specific application, or they can be used in a WebBroker or WebSnap program.

In this chapter I cannot cover every detail of IntraWeb—with 50 components installed on Delphi's palette and several module designers, it's a very large library. My plan is to cover its foundations, so that you can choose whether to use it for your forthcoming projects or for portions of those projects, as it fits best.

TIP *For documentation on IntraWeb, refer to the PDF manuals on the Delphi 7 Companion CD. If you cannot find them, these manuals are also available on Atozed Software's website for download. For support on IntraWeb, refer to the Borland newsgroups.*

NOTE *This chapter has had a special review by Chad Z. Hower, a.k.a. "Kudzu," the original author and project coordinator for Internet Direct (Indy; see Chapter 19, "Internet Programming: Sockets and Indy") and author of IntraWeb. Chad's areas of specialty include TCP/IP networking and programming, interprocess communication, distributed computing, Internet protocols, and object-oriented programming. When not programming, he likes to cycle, kayak, hike, downhill ski, drive, and do just about anything outdoors. Chad also posts free articles, programs, utilities, and other oddities at Kudzu World at* `http://www.Hower.org/Kudzu/`*. Chad is an American expatriate who currently spends his summers in St. Petersburg, Russia, and his winters in Limassol, Cyprus. Chad can be reached at* `cpub@Hower.org`.

This chapter covers the following topics:

- IntraWeb, web applications, and websites
- Using IntraWeb components
- Integrating with WebBroker and WebSnap
- Web database applications
- Using client-side components

Introducing IntraWeb

IntraWeb is a component library currently produced by Atozed Software (`www.atozedsoftware.com`). In Delphi 7 Professional and Enterprise, you can find the corresponding version of IntraWeb. The professional version can be used only in *Page mode*, as you'll see later in this chapter. Although Delphi 7 is the first version of the Borland IDE to include this set of components, IntraWeb has been around for several years; it has received appraisal and support, including the availability of several third-party components.

TIP *IntraWeb third-party components include IWChart by Steema (the makers of TeeChart), IWBold by Centillex (for integration with Bold), IWOpenSource, IWTranslator, IWDialogs, IWDataModulePool by Arcana, IW Component Pack by TMS, and IWGranPrimo by GranPrimo. You'll find an updated list of third parties on* `www.atozedsoftware.com`.

Although you don't have the source code for the core library (available on request and upon a specific payment), the IntraWeb architecture is fairly open, and the full source code of the components is readily available. IntraWeb is part of the Delphi standard installation now, but it is also available for Kylix. Written with care, IntraWeb applications can be fully cross-platform.

NOTE *In addition to the Delphi and Linux versions, C++ Builder and Java versions of IntraWeb are available. A .NET version is in the works and will probably become available along with a future Delphi for .NET version.*

As an owner of Delphi 7, you're entitled to receive the first significant update release (version 5.1) and can upgrade your license to a full IntraWeb Enterprise edition including upgrades and support from Atozed Software (see their website for details). More serious documentation (help and PDF files) will be available in this 5.1 upgrade.

From Websites to Web Applications

As I mentioned earlier, the idea behind IntraWeb is to build web applications rather than websites. When you work with WebBroker or WebSnap (covered in Chapter 20, "Web Programming with WebBroker and WebSnap"), you think in terms of web pages and page producers, and work closely at the HTML generation level. When you work with IntraWeb, you think in terms of components, their properties, and their event handlers, as you do in Delphi visual development.

For example, create a new IntraWeb application by selecting File ➢ New ➢ Other, moving to the IntraWeb page of the New Items dialog box, and choosing Stand Alone Application. In the following dialog box (which is part of Delphi, not of the IntraWeb wizard) you can choose an existing folder or enter a new one that will be created for you (I'm mentioning this because it is far from clear in the dialog). The resulting program has a project file and two different units (I'll cover this structure later).

For the moment, let's create an example (called IWSimpleApp in the book's source code). To build it, follow these steps:

1. Move to the main form of the program and add to it a button, an edit box, and a list box from the IW Standard page of the Components palette. That is, don't add the VCL components from the Standard page of the palette—instead, use the corresponding IntraWeb components: IWButton, IWEdit, and IWListbox.
2. Slightly modify their properties as follows:

   ```
   object IWButton1: TIWButton
     Caption = 'Add Item'
   end
   object IWEdit1: TIWEdit
     Text = 'four'
   end
   object IWListbox1: TIWListbox
     Items.Strings = (
       'one'
       'two'
       'three')
   end
   ```

3. Handle the `OnClick` event of the button by double-clicking on the component at design time as usual and writing this familiar code:

   ```
   procedure TformMain.IWButton1Click(Sender: TObject);
   begin
     IWListBox1.Items.Add (IWEdit1.Text);
   end;
   ```

That's all it takes to create a web-based application capable of adding text to a list box, as you can see in Figure 21.1 (which shows the final version of the program, with a couple more buttons). The important thing to notice when you run this program is that every time you click the button, the browser sends a new request to the application, which runs the Delphi event handler and produces a new HTML page based on the new status of the components on the form.

As you execute the application, you won't see the program output in the browser, but rather IntraWeb's controller form, shown in Figure 21.2. A stand-alone IntraWeb application, is a full-blown HTTP server, as you'll see in more detail in the next section. The form you see is managed by the `IWRun` call in the project file created by default in each stand-alone IntraWeb application. The debug form allows you to select a browser and run the application through it or copy the URL of the application to the Clipboard, so you can paste it within your browser. It's important to know that the application will by default use a random port number, which is different for each execution; so, you'll have to use a different URL every time. You can modify this behavior by selecting the server controller's designer (similar to a data module) and setting the `port` property. In the example I've used 8080 (one of the common HTTP ports), but any other value will do.

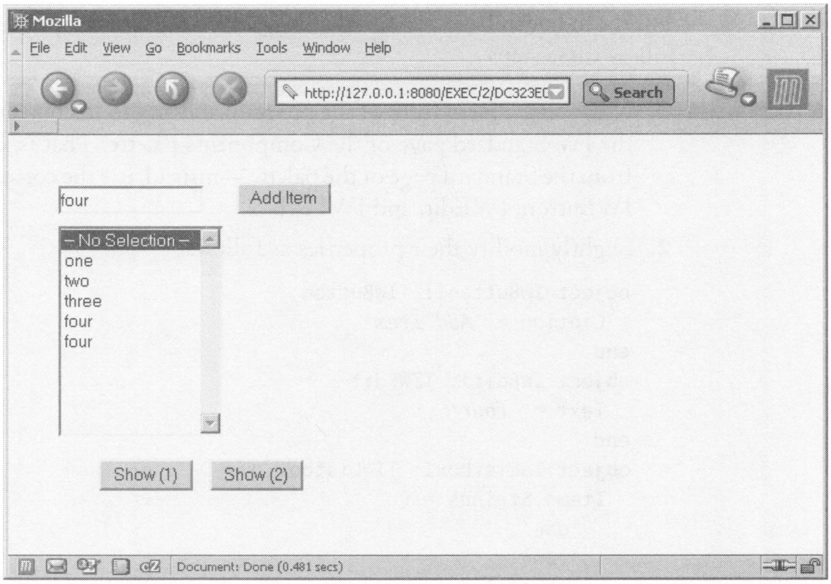
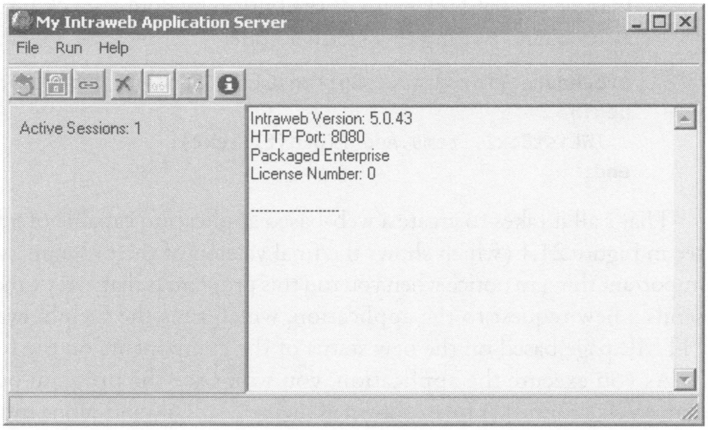

Mozilla
File Edit View Go Bookmarks Tools Window Help
http://127.0.0.1:8080/EXEC/2/DC323E0
Search
four
Add Item
-- No Selection --
one
two
three
four
four
Show (1)
Show (2)
Document: Done (0.481 secs)

FIGURE 21.1
The IWSimpleApp program in a browser


My Intraweb Application Server
File Run Help

```
Active Sessions: 1
Intraweb Version: 5.0.43
HTTP Port: 8080
Packaged Enterprise
License Number: 0

-------------------
```

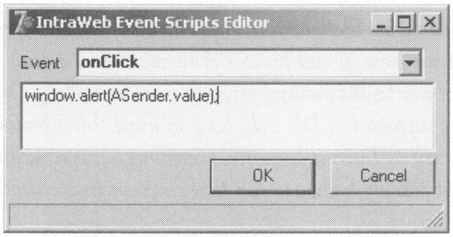
FIGURE 21.2
The controller form of a stand-alone IntraWeb application

IntraWeb code is mainly server side, but IntraWeb also generates JavaScript to control some application features; you can also execute extra code on the client side. You do so by using specific client-side IntraWeb components or by writing your own custom JavaScript code. As a comparison, the two buttons at the bottom of the form in the IWSimpleApp example show a message box using two different approaches.

The first of these two buttons (`IWButton2`) shows a message using a server-side event, with this Delphi code:

```
procedure TformMain.IWButton2Click(Sender: TObject);
var
  nItem: Integer;
begin
  nItem := IWListbox1.ItemIndex;
  if nItem >= 0 then
    WebApplication.ShowMessage (IWListBox1.Items [nItem])
  else
    WebApplication.ShowMessage ('No item selected');
end;
```

The second of these two buttons (`IWButton3`) uses JavaScript, which is inserted in the Delphi program by setting the proper JavaScript event handler in the special property editor for the `ScriptEvents` property:



```
window.alert(ASender.value);
```

A First Look Behind the Scenes

You have seen that creating an IntraWeb application is as simple as creating a Delphi form-based program: You place components on a form and handle their events. Of course, the effect is rather different, because the application runs in a browser. To give you a feel for what's going on, let's briefly look behind the scenes of this simple program. Doing so should help you understand the effect of setting the component properties and of working with them in general.

This is a browser-based program, so there is no better way to understand it than by looking at the HTML it sends to the browser. Open the source of the IWSimpleApp program's page (not listed here, because it would waste too much book space), and you'll notice it is divided into three main sections. The first is a list of styles (based on the HTTP `style` tag) with lines like the following:

```
.IWEDIT1CSS {position:absolute;left:40;top:40;z-index:100;
  font-style:normal;font-size:10pt;text-decoration:none;}
```

IntraWeb uses styles to determine not only the visual appearance of each component, such as its font and color, but also the component's position, using default absolute positioning. Each style is affected by a number of the IntraWeb component's properties, so you can easily experiment if you have some knowledge of style sheets. If you aren't familiar with style sheets, just use the properties and trust that IntraWeb will do its best to render the components in the web page.

The second block consists of JavaScript scripting. The main script block contains initialization code and the code of client-side event handlers for the components, like the following:

```
function IWBUTTON1_OnClick(ASender) {
  return SubmitClickConfirm('IWBUTTON1','', true, '');
}
```

This handler invokes the corresponding server-side code. If you've provided the JavaScript code directly in the IntraWeb application, as discussed earlier, you'll see this code:

```
function IWBUTTON3_onClick(ASender) {
  window.alert(ASender.value);
}
```

The scripting section of the page has also references to other files required by the browser and made available by IntraWeb. Some of these files are generic; others are tied to the specific browser: IntraWeb detects the browser being used and returns different JavaScript code and base JavaScript files.

NOTE *Because JavaScript is not identical on all browsers, IntraWeb supports only some of them, including all recent versions of Microsoft Internet Explorer, Netscape Navigator, and the open-source Mozilla (which I've used while writing this chapter). Opera has more limited JavaScript support, so by default if it is recognized IntraWeb will issue an error (depending on the* `SupportedBrowsers` *property of the controller). Opera can be used with the free Arcana components and will officially be supported in IW 5.1. Keep in mind that a browser can fake its identity: For example, Opera is often set to identify itself as Internet Explorer, preventing a proper identification to make it possible to use sites restricted to other browsers, but possibly leading to run time errors or inconsistencies.*

The third part of the generated HTML is the definition of the page structure. Inside the `body` tag is a `form` tag (on the same line) with the next action to be executed:

```
<form onsubmit="return FormDefaultSubmit();" name="SubmitForm"
  action="/EXEC/3/DC323E01B09C83224E57E240" method="POST">
```

The `form` tag hosts the specific user interface components, such as buttons and edit boxes:

```
<input type="TEXT" name="IWEDIT1" size="17" value="four"
  id="IWEDIT1" class="IWEDIT1CSS">
<input value="Add Item" name="IWBUTTON1" type="button"
  onclick="return IWBUTTON1_OnClick(this);"
  id="IWBUTTON1" class="IWBUTTON1CSS">
```

The form has also a few hidden components that IntraWeb uses to pass information back and forth. However, the URL is the most important way to pass information in IntraWeb; in the program it looks like this:

```
http://127.0.0.1:8080/EXEC/2/DC323E01B09C83224E57E240
```

The first part is the IP address and port used by the stand-alone IntraWeb application (it changes when you use a different architecture to deploy the program), followed by the `EXEC` command, a progressive request number, and a session ID. We'll get back to session later on, but suffice to say that IntraWeb uses a URL token instead of cookies to make its applications available regardless of browser settings. If you prefer, you can use cookies instead of URLs by changing the `TrackMode` property in the server controller.

WARNING *The version of IntraWeb that shipped with Delphi 7 had a bug involving cookies and some time zone settings. It has been fixed, and the patch is available as a free update on the Atozed software website.*

IntraWeb Architectures

Before I write more examples to demonstrate the use of other IntraWeb components available in Delphi 7, let's discuss another key element of IntraWeb: the different architectures you can use to create and deploy applications based on this library. You can create IntraWeb projects in Application mode (which accounts for all of the features of IntraWeb) or Page mode (which is a simplified version you can plug into existing Delphi WebBroker or WebSnap applications). Application mode applications can be deployed as ISAPI libraries, Apache modules, or by using IntraWeb Standalone mode (a variation of the Application mode architecture). Page mode programs can be deployed as any other WebBroker application (ISAPI, Apache module, CGI, etc.). IntraWeb features three different but partially overlapping architectures:

Standalone Mode Provides you with a custom web server, as in the first example you built. This is handy for debugging the application (because you can run it from the Delphi IDE and place breakpoints anywhere in the code). You can also use Standalone mode to deploy applications on internal networks (intranets) and to let users work in offline mode on their own computers, with a web interface. If you run a stand-alone IntraWeb program with the `-install` flag, it will run as a service, and the dialog box won't appear. Standalone mode gives you a way to deploy an Application mode IntraWeb program using IntraWeb itself as web server.

Application Mode Allows you to deploy an IntraWeb application on a commercial server, building an Apache module or an IIS library. Application mode includes session management and all the IntraWeb features and is the preferred way to deploy a scalable application for use on the Web. To be precise, Application mode IntraWeb programs can be deployed as stand-alone programs, ISAPI libraries, or Apache modules.

Page Mode Opens a way to integrate IntraWeb pages in WebBroker and WebSnap applications. You can add features to existing programs or rely on other technologies for portions of a dynamic site based on HTML pages, while providing the interactive portions with IntraWeb. Page mode is the only choice for using IntraWeb in CGI applications, but it lacks session-management features. Stand-alone IntraWeb servers do not support Page mode.

In the examples in the rest of the chapter, I'll use Standalone mode for simplicity and easier debugging, but I'll also cover Page mode support.

Building IntraWeb Applications

When you build an IntraWeb application, a number of components are available. For example, if you look at the IW Standard page of Delphi's Component Palette, you'll see an impressive list of core components, from the obvious button, check box, radio button, edit box, list box, memo, and so on to the intriguing tree view, menu, timer, grid, and link components. I won't list each component and describe its use with an example—I'd rather use some of the components in a few demos and underline the architecture of IntraWeb rather than specific details.

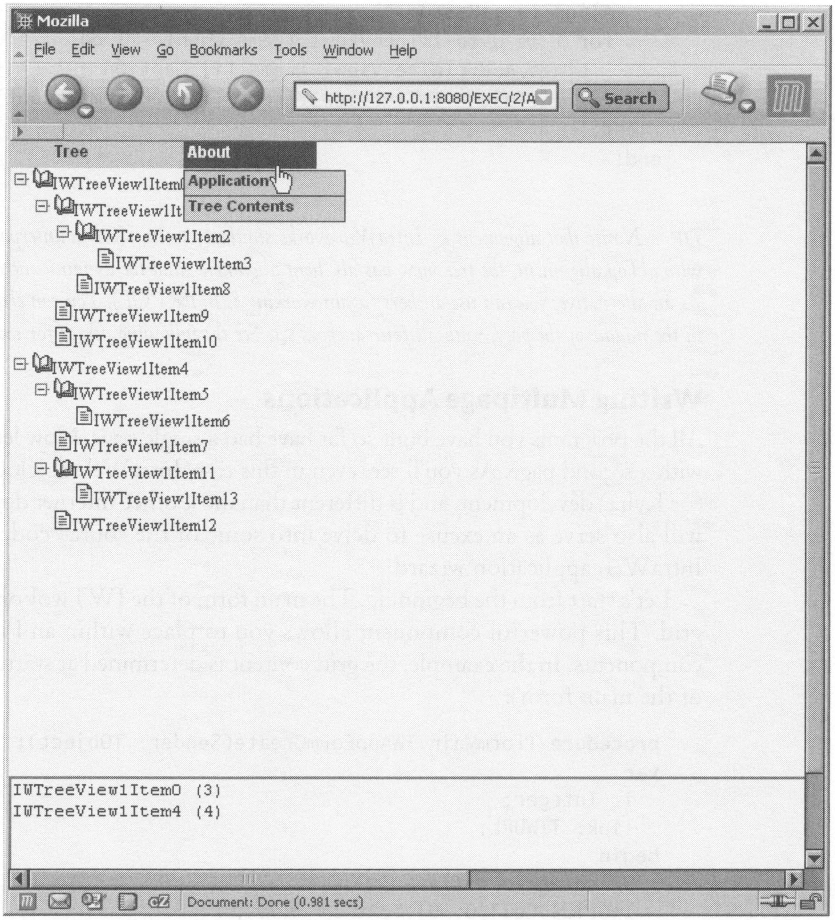
I've built an example (called IWTree) showcasing the menu and tree view components of IntraWeb but also featuring the creation of a component at run time. This handy component makes available in a dynamic menu the content of a standard Delphi menu, by referring its `AttachedMenu` property to a `TMenu` component:

```
object MainMenu1: TMainMenu
  object Tree1: TMenuItem
    object ExpandAll1: TMenuItem
    object CollapseAll1: TMenuItem
    object N1: TMenuItem
    object EnlargeFont1: TMenuItem
    object ReduceFont1: TMenuItem
  end
  object About1: TMenuItem
    object Application1: TMenuItem
    object TreeContents1: TMenuItem
  end
end
object IWMenu1: TIWMenu
  AttachedMenu = MainMenu1
  Orientation = iwOHorizontal
end
```

If the menu items handle the `OnClick` event in the code, they become links at run time. You can see an example of a menu in a browser in Figure 21.3. The example's second component is a tree view with a set of predefined nodes. This component has a lot of JavaScript code to let you expand and collapse nodes directly in the browser (with no need to call back the server). At the same time, the menu items allow the program to operate on the menu by expanding or collapsing nodes and changing the font. Here is the code for a couple of event handlers:

```
procedure TformTree.ExpandAll1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to IWTreeView1.Items.Count - 1 do
    IWTreeView1.Items [i].Expanded := True;
end;

procedure TformTree.EnlargeFont1Click(Sender: TObject);
begin
  IWTreeView1.Font.Size := IWTreeView1.Font.Size + 2;
end;
```

Thanks to the similarity of IntraWeb components to standard Delphi VCL components, the code is easy to read and understand.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://127.0.0.1:8080/EXEC/2/A
Search
Tree
About
Application
Tree Contents
IWTreeView1Item0
IWTreeView1Item1
IWTreeView1Item2
IWTreeView1Item3
IWTreeView1Item8
IWTreeView1Item9
IWTreeView1Item10
IWTreeView1Item4
IWTreeView1Item5
IWTreeView1Item6
IWTreeView1Item7
IWTreeView1Item11
IWTreeView1Item13
IWTreeView1Item12
Document: Done (0.981 secs)

```
IWTreeView1Item0 (3)
IWTreeView1Item4 (4)
```

FIGURE 21.3 The IWTree example features a menu, a tree view, and the dynamic creation of a memo component.

The menu has two submenus, which are slightly more complex. The first displays the application ID, which is an application execution/session ID. This identifier is available in the `AppID` property of the `WebApplication` global object. The second submenu, Tree Contents, shows a list of the first tree nodes of the main level along with the number of direct subnodes. What's interesting, though, is that the information is displayed in a memo component created at run time (see again Figure 21.3), exactly as you'll do in a VCL application:

```
procedure TformTree.TreeContents1Click(Sender: TObject);
var
  i: Integer;
begin
  with TIWMemo.Create(Self) do
  begin
    Parent := Self;
```

```
      Align := alBottom;
      for i := 0 to IWTreeView1.Items.Count - 1 do
        Lines.Add (IWTreeView1.Items [i].Caption + ' (' +
          IntToStr (IWTreeView1.Items [i].SubItems.Count) + ')');
    end;
  end;
```

TIP *Notice that alignment in IntraWeb works similarly to its VCL counterpart. For example, this program's menu with alTop alignment, the tree view has alClient alignment, and the dynamic memo is created with alBottom alignment. As an alternative, you can use anchors (again working as in the VCL): You can create bottom-right buttons, or components in the middle of the page, with all four anchors set. See the following demos for examples of this technique.*

Writing Multipage Applications

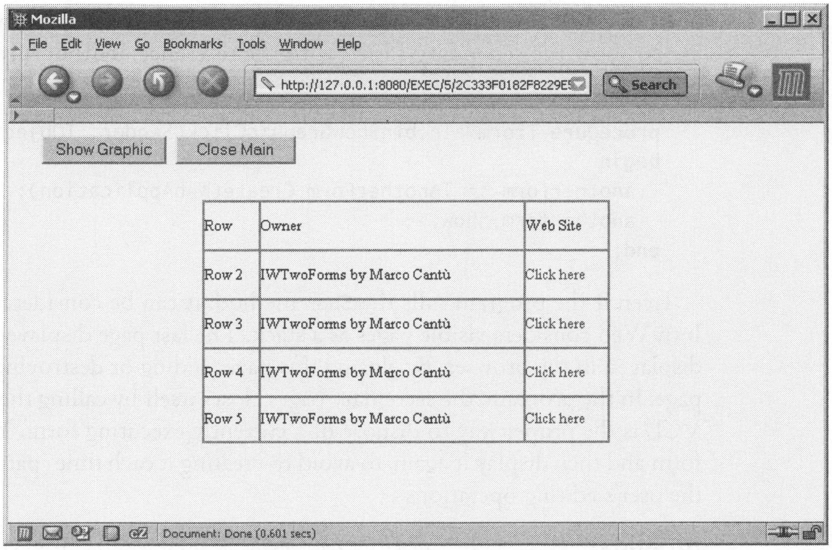
All the programs you have built so far have had a single page. Now let's create an IntraWeb application with a second page. As you'll see, even in this case, IntraWeb development resembles standard Delphi (or Kylix) development, and is different than most other Internet development libraries. This example will also serve as an excuse to delve into some of the source code automatically generated by the IntraWeb application wizard.

Let's start from the beginning. The main form of the IWTwoForms example features an IntraWeb grid. This powerful component allows you to place within an HTML grid both text and other components. In the example, the grid content is determined at startup (in the `OnCreate` event handler of the main form):

```
procedure TformMain.IWAppFormCreate(Sender: TObject);
var
  i: Integer;
  link: TIWURL;
begin
  // set grid titles
  IWGrid1.Cell[0, 0].Text := 'Row';
  IWGrid1.Cell[0, 1].Text := 'Owner';
  IWGrid1.Cell[0, 2].Text := 'Web Site';
  // set grid contents
  for i := 1 to IWGrid1.RowCount - 1 do
  begin
    IWGrid1.Cell [i,0].Text := 'Row ' + IntToStr (i+1);
    IWGrid1.Cell [i,1].Text := 'IWTwoForms by Marco Cantù';
    link := TIWURL.Create(Self);
    link.Text := 'Click here';
    link.URL := 'http://www.marcocantu.com';
    IWGrid1.Cell [i,2].Control := link;
  end;
end;
```

The effect of this code is shown in Figure 21.4. In addition to the output, there are a few interesting things to notice. First, the grid component uses Delphi anchors (all set to False) to generate code that keeps it centered in the page, even if a user resizes the browser window. Second, I've added an IWURL component to the third column, but you could add any other component (including buttons and edit boxes) to the grid.

FIGURE 21.4
The IWTwoForms example uses an IWGrid component, embedded text, and IWURL components.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://127.0.0.1:8080/EXEC/5/2C333F0182F8229E5
Search
Show Graphic
Close Main
Document: Done (0.601 secs)

Row	Owner	Web Site
Row 2	IWTwoForms by Marco Cantù	Click here
Row 3	IWTwoForms by Marco Cantù	Click here
Row 4	IWTwoForms by Marco Cantù	Click here
Row 5	IWTwoForms by Marco Cantù	Click here

The third and most important consideration is that an IWGrid is translated into an HTML gird, with or without a frame around it. Here is a snippet of the HTML generated for one of the grid rows:

```
<tr>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;">Row 2</font>
  </td>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;">IWTwoForms by Marco Cantù</font>
  </td>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;"></font>
    <a href="#" onclick="parent.LoadURL('http://www.marcocantu.com')"
      id="TIWURL1" name="TIWURL1"
      style="z-index:100;font-style:normal;font-size:10pt;text-decoration:none;">
      Click here</a>
  </td>
</tr>
```

TIP *In the previous listing, notice that the linked URL is activated via JavaScript, not with a direct link. This happens because all actions in IntraWeb allow for extra client-side operations, such as validations, checks, and submits. For example, if you set the* `Required` *property for a component, if the field is empty the data won't be submitted, and you'll see a JavaScript error message (customizable by setting the descriptive* `FriendlyName` *property).*

The core feature of the program is its ability to show a second page. To accomplish this, you first need to add a new IntraWeb page to the application, using the Application Form option on the IntraWeb page of Delphi's New Items dialog box (File ➢ New ➢ Other). Add to this page a few IntraWeb components, as usual, and then add to the main form a button or other control you'll use to show the secondary form (with the reference `anotherform` stored in a field of the main form):

```
procedure TformMain.btnShowGraphicClick(Sender: TObject);
begin
  anotherform := TAnotherForm.Create(WebApplication);
  anotherform.Show;
end;
```

Even if the program calls the `Show` method, it can be considered like a `ShowModal` call, because IntraWeb considers visible pages as a stack. The last page displayed is on the top of the stack and is displayed in the browser. By closing this page (hiding or destroying it), you re-display the previous page. In the program, the secondary pages closes itself by calling the `Release` method, which as in the VCL is the proper way to dispose of a currently executing form. You can also hide the secondary form and then display it again, to avoid re-creating it each time (particularly if doing so implies losing the user's editing operations).

WARNING *In the program I added a Close button to the main form. It should not call* `Release`*, but rather should invoke the* `WebApplication` *object's* `Terminate` *method, passing the output message, as in* `WebApplication.Terminate('Goodbye!')`*. The demo uses an alternative call:* `TerminateAndRedirect`.

Now that you have seen how to create an IntraWeb application with two forms, let's briefly examine how IntraWeb creates the main form. The relevant code, generated by the IntraWeb wizard as you create a new program, is in the project file:

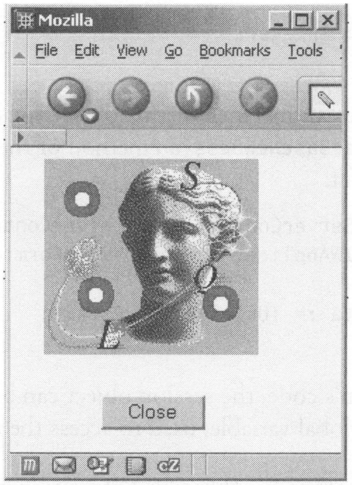
```
begin
  IWRun(TFormMain, TIWServerController);
```

This is different from Delphi's standard project file, because it calls a global function instead of applying a method to a global object representing the application. The effect, though, is quite similar. The two parameters are the classes of the main form and of the IntraWeb controller, which handle sessions and other features as you'll see in a while.

The secondary form of the IWTwoForms example shows another interesting feature of IntraWeb: its extensive graphics support. The form has a graphical component with the classic Delphi Athena image. This is accomplished by loading a bitmap into the an IWImage component: IntraWeb converts the bitmap into a JPEG, stores it in a cache folder created under the application folder, and returns a reference to it, with the following HTML:

```
<img src="/cache/JPG1.tmp" name="IWIMAGE1" border="0" width="153" height="139">
```

The extra feature provided by IntraWeb and exploited by the program is that a user can click on the image with the mouse to modify the image by launching server-side code. In this program, the effect is to draw small green circles.



This effect is obtained with the following code:

```
procedure Tanotherform.IWImage1MouseDown(ASender: TObject;
  const AX, AY: Integer);
var
  aCanvas: TCanvas;
begin
  aCanvas := IWImage1.Picture.Bitmap.Canvas;
  aCanvas.Pen.Width := 8;
  aCanvas.Pen.Color := clGreen;
  aCanvas.Ellipse(Ax - 10, Ay - 10, Ax + 10, Ay + 10);
end;
```

WARNING *The painting operation takes place on the bitmap canvas. Do not try to use the Image canvas (as you can do with the VCL's TImage component), and do not try to use a JPEG in the first place, or you'll see either no effect or a run-time error.*

Sessions Management

If you've done any web programming, you know that session management is a complex issue. IntraWeb provides predefined session management and simplifies the way you work with sessions. If you need session data for a specific form, all you have to do is add a field to that form. The IntraWeb forms and their components have an instance for each user session. For example, in the IWSession example, I've added to the form a field called `FormCount`. As a contrast, I've also declared a global unit variable called `GlobalCount`, which is shared by all the instances (or sessions) of the application.

To increase your control over session data and let multiple forms share it, you can customize the `TUserSession` class that the IntraWeb Application Wizard places in the ServerController unit. In the IWSession example, I've customized the class as follows:

```
type
  TUserSession = class
  public
    UserCount: Integer;
  end;
```

IntraWeb creates an instance of this object for each new session, as you can see in the `IWServerControllerBaseNewSession` method of the `TIWServerController` class in the default ServerController unit:

```
procedure TIWServerController.IWServerControllerBaseNewSession(
  ASession: TIWApplication; var VMainForm: TIWAppForm);
begin
  ASession.Data := TUserSession.Create;
end;
```

In an application's code, the session object can be referenced by accessing the Data field of the `RWebApplication` global variable, used to access the current user's session.

NOTE *`RWebApplication` is a `threadvar` variable, defined in the IWInit unit. It gives you access to the session data in a thread-safe way: you need to take special care to access it even in a multi-threading environment. This variable can be used outside of a form or control (which are natively session-based), which is why it is primarily used inside data modules, global routines, and non-IntraWeb classes.*

Again, the default ServerController unit provides a helper function you can use:

```
function UserSession: TUserSession;
begin
  Result := TUserSession(RWebApplication.Data);
end;
```

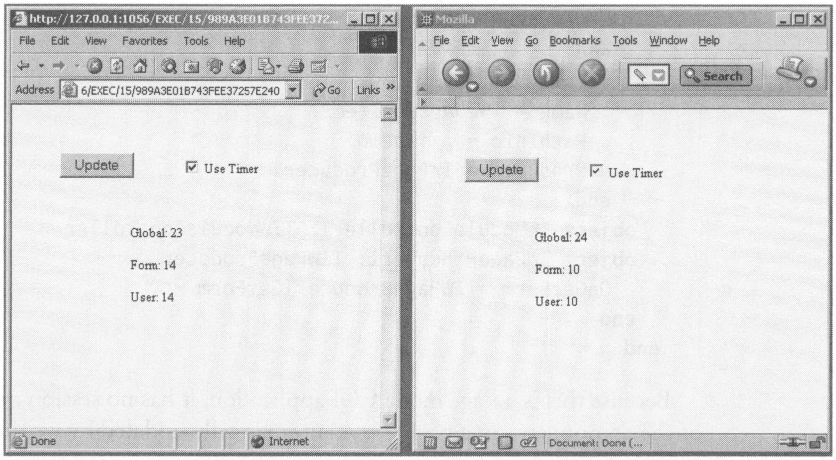
Because most of this code is generated for you, after adding data to the `TUserSession` class you simply use it through the `UserSession` function, as in the following code extracted from the IWSession example. When you click a button, the program increases several counters (one global and two session-specific) and shows their values in labels:

```
procedure TformMain.IWButton1Click(Sender: TObject);
begin
  InterlockedIncrement (GlobalCount);
  Inc (FormCount);

  Inc (UserSession.UserCount);

  IWLabel1.Text := 'Global: ' + IntToStr (GlobalCount);
  IWLabel2.Text := 'Form: ' + IntToStr (FormCount);
  IWLabel3.Text := 'User: ' + IntToStr (UserSession.UserCount);
end;
```

Notice that the program uses Windows' `InterlockedIncrement` call to avoid concurrent access to the global shared variable by multiple threads. Alternative approaches include using a critical section or Indy's `TidThreadSafeInteger` (found in the IdThreadsafe unit).

Figure 21.5 shows the output of the program (with two sessions running in two different browsers). The program has also a check box that activates a timer. Odd as it sounds, in an IntraWeb application, timers work almost the same as in Windows. When the timer interval expires, code is executed. Over the Web, this means refreshing the page by triggering a refresh in the JavaScript code:

```
IWTIMER1=setTimeout('SubmitClick("IWTIMER1","", false)',5000);
```


http://127.0.0.1:1056/EXEC/15/989A3E01B743FEE372...
File Edit View Favorites Tools Help
Address
6/EXEC/15/989A3E01B743FEE37257E240
Go
Links
Update
☑ Use Timer
Done
Internet
Mozilla
File Edit View Go Bookmarks Tools Window Help
Search
Update
☑ Use Timer
Document: Done (...

Global: 23
Form: 14
User: 14

Global: 24
Form: 10
User: 10

FIGURE 21.5 The IWSession application has both session-specific and global counters, as you can see by running two sessions in two different browsers (or even in the same browser).

Integrating with WebBroker (and WebSnap)

Up to now, you have built stand-alone IntraWeb applications. When you create an IntraWeb application in a library to be deployed on IIS or Apache, you are basically in the same situation. Things change considerably, however, if you want to use IntraWeb Page mode, which is integrated in an IntraWeb page in a WebBroker (or WebSnap) Delphi application.

The bridge between the two worlds is the IWPageProducer component. This component hooks to a WebBroker action like any other page producer component and has a special event you can use to create and return an IntraWeb form:

```
procedure TWebModule1.IWPageProducer1GetForm(ASender: TIWPageProducer;
  AWebApplication: TIWApplication; var VForm: TIWPageForm);
begin
  VForm := TformMain.Create(AWebApplication);
end;
```

With this single line of code (plus the addition of an IWModuleController component in the web module), the WebBroker application can embed an IntraWeb page, as the CgiIntra program does. The IWModuleController component provides *core services* for IntraWeb support. A component of this type must exist in every project for IntraWeb to work properly.

WARNING *The release that comes with Delphi 7 has a problem with Delphi's Web App Debugger and the IWModuleController component. This issue has been fixed and is a free update.*

Here is a summary of the DFM of the example program's web module:

```
object WebModule1: TWebModule1
  Actions = <
    item
      Default = True
      Name = 'WebActionItem1'
      PathInfo = '/show'
      OnAction = WebModule1WebActionItem1Action
    end
    item
      Name = 'WebActionItem2'
      PathInfo = '/iwdemo'
      Producer = IWPageProducer1
    end>
  object IWModuleController1: TIWModuleController
  object IWPageProducer1: TIWPageProducer
    OnGetForm = IWPageProducer1GetForm
  end
end
```

Because this is a Page mode CGI application, it has no session management. Moreover, the status of the components in a page is not automatically updated by writing event handlers, as in a standard IntraWeb program. To accomplish the same effect you need to write specific code to handle further parameters of the HTTP request. It should be clear even from this simple example that Page mode does less for you than Application mode, but it's more flexible. In particular, IntraWeb Page mode allows you to add visual RAD design capabilities to your WebBroker and WebSnap applications.

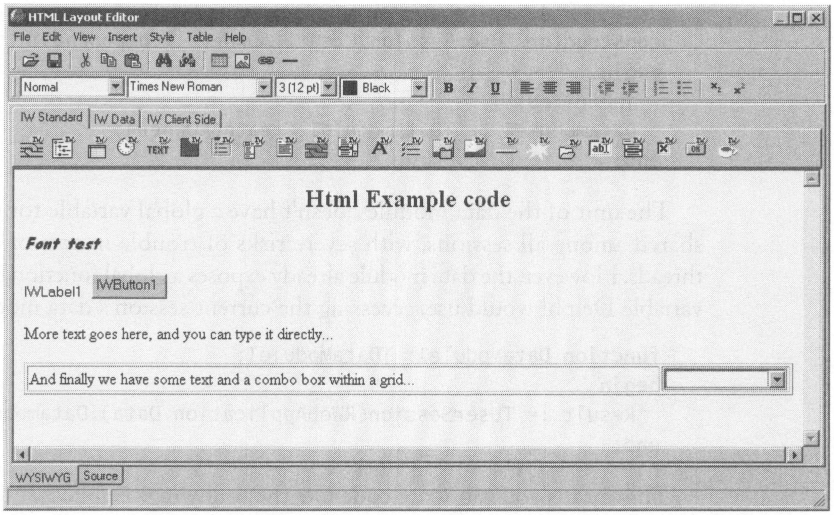
Controlling the Layout

The CgiIntra program features another interesting technology available in IntraWeb: the definition of a custom layout based on HTML. (That topic isn't really related, because HTML layouts work also in Application mode—but I've happened to use these two techniques in a single example.) In the programs built so far, the resulting page is the mapping of a series of components placed at design time on a form, in which you can use properties to modify the resulting HTML. But what if you want to embed a data-entry form within a complex HTML page? Building the entire page contents with IntraWeb components is awkward, even if you can use the IWText component to embed a custom piece of HTML within an IntraWeb page.

The alternative approach is represented by the use of IntraWeb's layout managers. In IntraWeb you invariably use a layout manager; the default is the IWLayoutMgrForm component. The other two alternatives are IWTemplateProcessorHTML for working with an external HTML template file and IWLayoutMgrHTML for working with internal HTML.

This second component includes a powerful HTML editor you can use to prepare the generic HTML as well as embed the required IntraWeb components (something you'll have to do manually with an external HTML editor). Moreover, as you select an IntraWeb component from this editor (which is activated by double-clicking on an IWLayoutMgrHTML component), you'll be able to use Delphi's Object Inspector to customize the component's properties. As you can see in Figure 21.6, the HTML Layout Editor available in IntraWeb is a powerful visual HTML editor; the HTML text it generates is available in a separate page. (The HTML editor will be improved in a coming upgrade, and a few quirks will be fixed.)

FIGURE 21.6
IntraWeb's HTML Layout Editor is a full-blown visual HTML editor.



Html Example code

Font test

More text goes here, and you can type it directly...

And finally we have some text and a combo box within a grid...

In the generated HTML, the HTML defines the structure of the page. The components are marked only with a special tag based on curly braces, as in the following of the example:

```
<P> {%IWLabel1%} {%IWButton1%}</P>
```

TIP *Notice that when you're using HTML the components don't use absolute positioning but flow along with the HTML. Thus the form becomes only a component holder, because the size and position of the form's components are ignored.*

Needless to say, the HTML you see in the visual designer of the HTML Layout Editor corresponds almost perfectly to the HTML you'll see when running the program in a browser.

Web Database Applications

As in Delphi's libraries, a significant portion of IntraWeb's available controls relates to the development of database applications. The IntraWeb Application Wizard has a version that allows you to create an application with a data module—a good starting point for the development of a database application. In such a case, the application's predefined code creates an instance of the data module for each session, saving it in the session's data.

Here is the predefined `TUserSession` class (and its constructor) for an IntraWeb application with a data module:

```
type
  TUserSession = class(TComponent)
  public
    DataModule1: TDataModule1;
    constructor Create(AOwner: TComponent); override;
  end;

constructor TUserSession.Create(AOwner: TComponent);
begin
  inherited;
  Datamodule1 := TDatamodule1.Create(AOwner);
end;
```

The unit of the data module doesn't have a global variable for it; if it did, all the data would be shared among all sessions, with severe risks of trouble in case of concurrent requests in multiple threads. However, the data module already exposes a global function having the same name as the global variable Delphi would use, accessing the current session's data module:

```
function DataModule1: TDataModule1;
begin
  Result := TUserSession(RWebApplication.Data).Datamodule1;
end;
```

This means you can write code like the following:

```
DataModule1.SimpleDataSet1
```

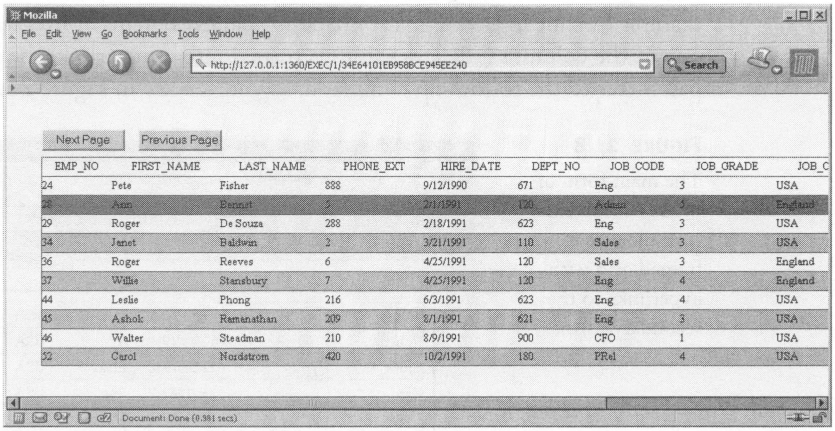
But instead of accessing a global data module, you are using the current session's data module.

In the first sample program featuring database data, called IWScrollData, I've added to the data module a SimpleDataSet component and to the main form an IWDBGrid component with the following configuration:

```
object IWDBGrid1: TIWDBGrid
  Anchors = [akLeft, akTop, akRight, akBottom]
  BorderSize = 1
  CellPadding = 0
  CellSpacing = 0
  Lines = tlRows
  UseFrame = False
  DataSource = DataSource1
  FromStart = False
  Options = [dgShowTitles]
  RowAlternateColor = clSilver
  RowLimit = 10
  RowCurrentColor = clTeal
end
```

The most important settings are the removal of a frame hosting the control with its own scroll bars (the `UseFrame` property), the fact that the data is displayed form the current data set position (the `FromStart` property), and the number of rows to be displayed in the browser (the `RowLimit` property). In the user interface, I've removed vertical lines and colored alternate rows. I also had to set up a color for the current row (the `RowCurrentColor` property); otherwise the alternate colors won't appear to work properly, since the current row is the same color as the background rows, regardless of its position (set the `RowCurrentColor` property to clNone to see what I mean). These settings produce the effect you can see in Figure 21.7 or by running the IWScrollData example.

FIGURE 21.7
The data-aware grid of the IWScrollData example


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://127.0.0.1:1360/EXEC/1/34E64101EB958BCE945EE240
Search
Next Page
Previous Page
Document: Done (0.981 secs)

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE	DEPT_NO	JOB_CODE	JOB_GRADE	JOB_C
24	Pete	Fisher	888	9/12/1990	671	Eng	3	USA
28	Ann	Bennet	5	2/1/1991	120	Admin	5	England
29	Roger	De Souza	288	2/18/1991	623	Eng	3	USA
34	Janet	Baldwin	2	3/21/1991	110	Sales	3	USA
36	Roger	Reeves	6	4/25/1991	120	Sales	3	England
37	Willie	Stansbury	7	4/25/1991	120	Eng	4	England
44	Leslie	Phong	216	6/3/1991	623	Eng	4	USA
45	Ashok	Ramanathan	209	8/1/1991	621	Eng	3	USA
46	Walter	Steadman	210	8/9/1991	900	CFO	1	USA
52	Carol	Nordstrom	420	10/2/1991	180	PRel	4	USA

The program opens the data set when the form is created, using the data set hooked to the current data source:

```
procedure TformMain.IWAppFormCreate(Sender: TObject);
begin
  DataSource1.DataSet.Open;
end;
```

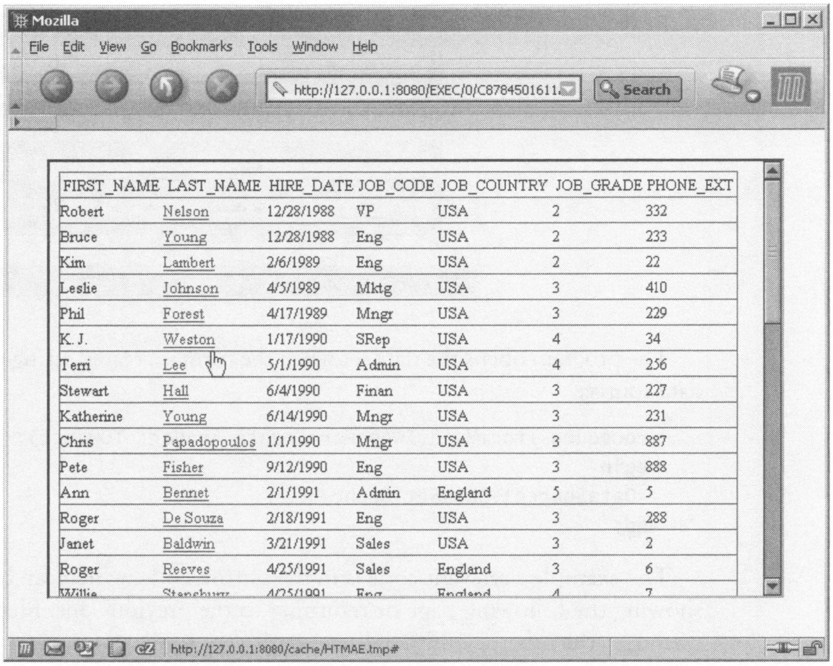
The example's relevant code is in the button code, which can be used to move through the data showing the following page or returning to the previous one. Here is the code for one of the two methods (the other is omitted, because it's very similar):

```
procedure TformMain.btnNextClick(Sender: TObject);
var
  i: Integer;
begin
  nPos := nPos + 10;
  if nPos > DataSource1.DataSet.RecordCount - 10 then
    nPos := DataSource1.DataSet.RecordCount - 10;
  DataSource1.DataSet.First;
  for i := 0 to nPos do
    DataSource1.DataSet.Next;
end;
```

Linking to Details

The grid of the IWScrollData example shows a single page of a table's data; buttons let you scroll up and down the pages. An alternative grid style in IntraWeb is offered by framed grids, which can move larger amounts of data to the web browser within a screen area of a fixed size using a frame and an internal scroll bar, as a Delphi ScrollBox control does. This is demonstrated by the IWGridDemo example.

The example customizes the grid in a second powerful way: It sets the `Columns` collection property of the grid. This setting allows you to fine-tune the output and behavior of specific columns, for example by showing hyperlinks or handling clicks on items or title cells. In the IWGridDemo example, one of the columns (the last name) is turned into a hyperlink; the employee number is passed as a parameter to the follow-up command, as you can see in Figure 21.8.


Mozilla
File
Edit
View
Go
Bookmarks
Tools
Window
Help
http://127.0.0.1:8080/EXEC/0/C8784501611
Search
http://127.0.0.1:8080/cache/HTMAE.tmp#

FIRST_NAME	LAST_NAME	HIRE_DATE	JOB_CODE	JOB_COUNTRY	JOB_GRADE	PHONE_EXT
Robert	Nelson	12/28/1988	VP	USA	2	332
Bruce	Young	12/28/1988	Eng	USA	2	233
Kim	Lambert	2/6/1989	Eng	USA	2	22
Leslie	Johnson	4/5/1989	Mktg	USA	3	410
Phil	Forest	4/17/1989	Mngr	USA	3	229
K. J.	Weston	1/17/1990	SRep	USA	4	34
Terri	Lee	5/1/1990	Admin	USA	4	256
Stewart	Hall	6/4/1990	Finan	USA	3	227
Katherine	Young	6/14/1990	Mngr	USA	3	231
Chris	Papadopoulos	1/1/1990	Mngr	USA	3	887
Pete	Fisher	9/12/1990	Eng	USA	3	888
Ann	Bennet	2/1/1991	Admin	England	5	5
Roger	De Souza	2/18/1991	Eng	USA	3	288
Janet	Baldwin	3/21/1991	Sales	USA	3	2
Roger	Reeves	4/25/1991	Sales	England	3	6
Willie	Stansbury	4/25/1991	Eng	England	4	7

FIGURE 21.8
The main form of the IWGridDemo example uses a framed grid with hyperlinks to the secondary form.

Listing 21.1 shows a summary of the grid's key properties. Notice in particular the last name column, which as mentioned has a linked field (which turns the cell's text into a hyperlink) and an event handler responding to its selection. In this method, the program creates a secondary form in which a user can edit the data:

```
procedure TGridForm.IWDBGrid1Columns1Click(ASender: TObject;
  const AValue: String);
begin
  with TRecordForm.Create (WebApplication) do
  begin
    StartID := AValue;
    Show;
  end;
end;
```

LISTING 21.1: PROPERTIES OF THE IWDBGRID IN THE IWGRIDDEMO EXAMPLE

```
object IWDBGrid1: TIWDBGrid
  Anchors = [akLeft, akTop, akRight, akBottom]
  UseFrame = True
  UseWidth = True
  Columns = <
    item
      Alignment = taLeftJustify
      BGColor = clNone
      DoSubmitValidation = True
      Font.Color = clNone
      Font.Enabled = True
      Font.Size = 10
      Font.Style = []
      Header = False
      Height = '0'
      VAlign = vaMiddle
      Visible = True
      Width = '0'
      Wrap = False
      BlobCharLimit = 0
      CompareHighlight = hcNone
      DataField = 'FIRST_NAME'
      Title.Alignment = taCenter
      Title.BGColor = clNone
      Title.DoSubmitValidation = True
      Title.Font.Color = clNone
      Title.Font.Enabled = True
      Title.Font.Size = 10
      Title.Font.Style = []
      Title.Header = False
      Title.Height = '0'
      Title.Text = 'FIRST_NAME'
      Title.VAlign = vaMiddle
      Title.Visible = True
      Title.Width = '0'
      Title.Wrap = False
    end
    item
      DataField = 'LAST_NAME'
      LinkField = 'EMP_NO'
      OnClick = IWDBGrid1Columns1Click
    end
    item
      DataField = 'HIRE_DATE'
    end
    item
      DataField = 'JOB_CODE'
    end
```

```
        item
          DataField = 'JOB_COUNTRY'
        end
        item
          DataField = 'JOB_GRADE'
        end
        item
          DataField = 'PHONE_EXT'
        end>
      DataSource = DataSource1
      Options = [dgShowTitles]
    end
```

By setting the second form's `StartID` property, you can locate the proper record:

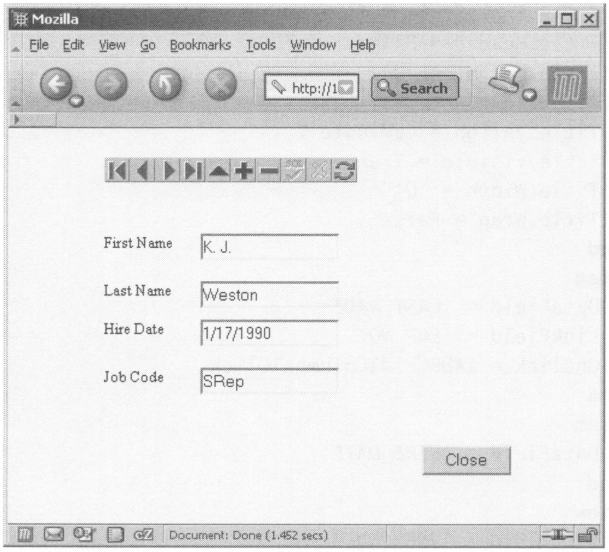
```
procedure TRecordForm.SetStartID(const Value: string);
begin
  FStartID := Value;
  DataSource1.DataSet.Locate('EMP_NO', Value, []);
end;
```

TIP *The IWDBGrid columns have also an* `OnTitleClick` *event you can handle to sort the data or perform other operations on the column.*

The secondary form is hooked to the same data module as the main form. So, after the database data is updated, you can see it in the grid (but the updates are kept only in memory, because the program doesn't have an `ApplyUpdates` call). The secondary form uses a few edit controls and a navigator, provided by IntraWeb. You can see this form at run time in Figure 21.9.

FIGURE 21.9

The secondary form of the IWGridDemo example allows a user to edit the data and navigate through records.



First Name: K. J.
Last Name: Weston
Hire Date: 1/17/1990
Job Code: SRep

Document: Done (1.452 secs)

Moving Data to the Client Side

Regardless of how you use it, the IWDBGrid component produces HTML with the database data embedded in the cells, but it cannot work on the data on the client side. A different component (or a set of IntraWeb components) allows you to follow a different model. The data is sent to the browser in a custom format, and the JavaScript code on the browser populates a grid and operates on the data, moving from record to record without asking more data to the server.

NOTE *This architecture is similar to Delphi's native Internet Express architecture, which I'll cover in Chapter 22 ("Using XML Technologies").*

You can use several IntraWeb components for a client-side application, but these are the most important ones:

IWClientSideDataSet An in-memory dataset you define by setting the `ColumnNames` and `Data` properties within your program's code. In future updates, you will be able to edit client-side data, sort it, filter it, define master-detail structures, and more.

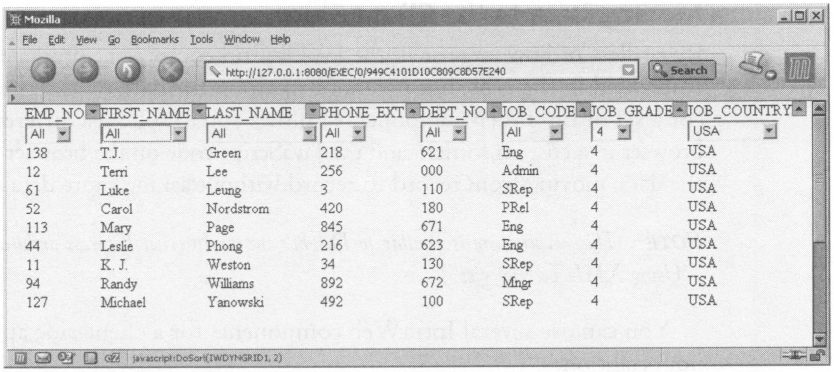
IWClientSideDataSetDBLink A data provider you can connect to any Delphi dataset, connecting it with the `DataSource` property.

IWDynGrid A dynamic grid component connected to one of the two previous components using the `Data` property. This component moves all the data to the browser and can operate on it on the client via JavaScript.

There are other client-side components in IntraWeb, such as IWCSLabel, IWCSNavigator, and IWDynamicChart (which works only with Internet Explorer). As an example of using this approach, I've built the IWClientGrid example. The program has little code, because there is a lot available in the components being used. Here are the core elements of its main form:

```
object formMain: TformMain
  SupportedBrowsers = [brIE, brNetscape6]
  OnCreate = IWAppFormCreate
  object IWDynGrid1: TIWDynGrid
    Align = alClient
    Data = IWClientSideDatasetDBLink1
  end
  object DataSource1: TDataSource
    Left = 72
    Top = 88
  end
  object IWClientSideDatasetDBLink1: TIWClientSideDatasetDBLink
    DataSource = DataSource1
  end
end
```

The dataset from the data module is hooked to the DataSource when the form is created. The resulting grid, shown in Figure 21.10, allows you to sort the data on any cell (using the small arrow *after* the column title) and filter the data displayed on one of the field's possible values. In the figure, for example, you can sort the employee data by last name and filter it by country and job grade.



Mozilla
File Edit View Go Bookmarks Tools Window Help
http://127.0.0.1:8080/EXEC/0/949C4101D10C809C8D57E240
Search
javascript:DoSort(IWDYNGRID1, 2)

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	DEPT_NO	JOB_CODE	JOB_GRADE	JOB_COUNTRY
All	All	All	All	All	All	4	USA
138	T.J.	Green	218	621	Eng	4	USA
12	Terri	Lee	256	000	Admin	4	USA
61	Luke	Leung	3	110	SRep	4	USA
52	Carol	Nordstrom	420	180	PRel	4	USA
113	Mary	Page	845	671	Eng	4	USA
44	Leslie	Phong	216	623	Eng	4	USA
11	K. J.	Weston	34	130	SRep	4	USA
94	Randy	Williams	892	672	Mngr	4	USA
127	Michael	Yanowski	492	100	SRep	4	USA

FIGURE 21.10
The grid of the IWClientGrid example supports custom sorting and filtering without re-fetching the data on the web server.

This functionality is possible because the data is moved to the browser within the JavaScript code. Here is a snippet of one of the scripts embedded in the page's HTML:

```
<script language="Javascript1.2">
var IWDYNGRID1_TitleCaptions =
  ["EMP_NO","FIRST_NAME","LAST_NAME","PHONE_EXT",
  "DEPT_NO","JOB_CODE","JOB_GRADE","JOB_COUNTRY"];
var IWDYNGRID1_CellValues = new Array();
IWDYNGRID1_CellValues[0] = [2,'Robert','Nelson','332','600','VP',2,'USA'];
IWDYNGRID1_CellValues[1] = [4,'Bruce','Young','233','621','Eng',2,'USA'];
IWDYNGRID1_CellValues[2] = [5,'Kim','Lambert','22','130','Eng',2,'USA'];
IWDYNGRID1_CellValues[3] = [8,'Leslie','Johnson','410','180','Mktg',3,'USA'];
IWDYNGRID1_CellValues[4] = [9,'Phil','Forest','229','622','Mngr',3,'USA'];
```

NOTE *The reason to use this JavaScript-based approach, instead of an XML-based approach featured by other similar technologies, is that only Internet Explorer has support for* XML data islands. *Mozilla and Netscape lack this feature and have limited XML support in general. Mimicking it in JavaScript, as Internet Explorer does, is very expensive at run time.*

What's Next?

This chapter's description of IntraWeb's features has been far from complete, but my aim was mainly to let you evaluate this technology so you can choose whether to use it in your forthcoming Delphi projects. IntraWeb is so powerful that you now have a good reason to build web applications in Delphi, instead of resorting to other development tools.

You can read much more about IntraWeb in the documentation, found on the Delphi Companion CD (not on the main CD). Delphi's default installation includes the IntraWeb demos, including the extensive Features example that shows at once most of the features of this component library. Also refer to the IntraWeb website (`www.atozedsoftware.com`) for updates and further documentation and examples.

In this book, we have another alternative web development approach to cover, based on XML and XSLT. Chapter 22 is devoted to a complete roundup of XML-related technologies from the Delphi perspective. So, we'll have another chance to cover web development in Delphi, using one of the techniques I like best—but that's also one of the most complex.

Chapter 22

Using XML Technologies

BUILDING APPLICATIONS FOR THE Internet means using protocols and creating browser-based user interfaces, as you've done in the preceding two chapters, but also opens an opportunity for exchanging business documents electronically. The emerging standards for this type of activity center around the XML document format and include the SOAP transmission protocol, XML schemas for the validation of documents, and XSL for rendering documents as HTML.

In this chapter, I'll discuss the core XML technologies and the extensive support Delphi has offered for them since version 6. Because XML knowledge is far from widespread, I'll provide a little introduction about each technology, but you should refer to books specifically devoted to them to learn more. In Chapter 23, I'll focus specifically on web services and SOAP.

This chapter covers the following topics:

- Introducing XML: Extensible Markup Language
- Working with an XML DOM
- Delphi and XML: interfaces and mapping
- Parsing XML documents with SAX
- Internet Express
- Using XSLT
- XSL in WebSnap

Introducing XML

Extensible Markup Language (XML) is a simplified version of SGML and is getting a lot of attention in the IT world. XML is a *markup language,* meaning it uses symbols to describe its own content—in this case, *tags* consisting of specially defined text enclosed in angle brackets. It is *extensible* because it allows for free markers (in contrast, for example, to HTML, which has predefined markers). The XML language is a standard promoted by the World Wide Web Consortium (W3C). The XML Recommendation is at `www.w3.org/TR/REC-xml`.

XML has been touted as the ASCII of the year 2000, to indicate a simple and widespread technology and also to indicate that an XML document is a plain-text file (optionally with Unicode characters instead of plain ASCII text). The important characteristic of XML is that it is descriptive, because every tag has an almost human-readable name. Here is an example, in case you've never seen an XML document:

```
<book>
  <title>Mastering Delphi 7</title>
  <author>Cantu</author>
  <publisher>Sybex</publisher>
</book>
```

XML has a few disadvantages I want to underline from the beginning. The biggest is that without a formal description, a document is worth little. If you want to exchange documents with another company, you must agree on what each tag means and also on the semantic meaning of the content. (For example, when you have a quantity, you have to agree on the measurement system or include it in the document.) Another disadvantage is that XML documents are much larger than other formats; using strings for numbers, for example, is far from efficient, and the repeated opening and closing tags eat up a lot of space. The good news is that XML compresses well, for the same reason.

Core XML Syntax

A few technical elements of XML are worth knowing before we discuss its usage in Delphi. Here is a short summary of the key elements of the XML syntax:

- White space (including the space character, carriage return, line feed, and tabs) is generally ignored (as in an HTML document). It is important to format an XML document to make it readable by a human being, but your programs won't care much.
- You can add comments within `<!--` and `-->` markers, which are basically ignored by XML processors. There are also directives and processing instructions, enclosed within `<?` and `?>` markers.
- There a few special or reserved characters you cannot use in the text. The only two symbols you can *never* use are the less-than character (or left angle bracket, <, used to delimit a marker), which is replaced by `&lt;`, and the ampersand character (&), which is replaced by `&amp;`. Other optional special characters are `&gt;` for the greater-than symbol (right angle bracket, >), `&apos;` for the single quote ('), and `&quot;` for the double quote (").
- To add non-XML content (for example, binary information or a script), you can use a CDATA section, enclosed within `<![CDATA[` and `]]>`.
- All markers are enclosed by angle brackets, `<` and `>`. Markers are case sensitive (in contrast to HTML).
- For each opening marker, you must have a matching closing marker, denoted by an initial slash character:

  ```
  <node>value</node>
  ```

- Markers must not overlap—they must be *properly nested,* as in the first line here (the second line is not correct):

  ```
  <node>xx <nested> yy</nested> </node>  // OK
  <node>xx <nested> yy</node> </nested>  // WRONG
  ```

- If a marker has no content (but its presence is important), you can replace the opening and closing markers with a single marker that includes a final (*trailing*) slash: `<node/>`.
- Markers can have attributes, using multiple attribute names followed by a value enclosed within quotes:

  ```
  <node attrib1="aaa">
  ```

- Any XML node can have multiple attributes, multiple embedded tags, and only one block of text, representing the value of the node. It is common practice for XML nodes to have either a textual value or embedded tags, and not both. Here is an example of the full syntax of a node:

  ```
  <node attrib1="aaa" attrib2="bbb">
    value1
    <child1>
      value2
    </child1>
  </node>
  ```

- A node can have multiple child nodes with the same tag (tags need not be unique). Attribute names are unique for each node.

Well-Formed XML

The elements discussed in the previous section define the syntax of an XML document, but they are not enough. An XML document is considered syntactically correct, or *well formed,* if it follows a few extra rules. Notice that this type of check doesn't guarantee that the content of the document is meaningful—only that the tags are properly laid out.

Each document should have a prologue indicating that it is indeed an XML document, which version of XML it complies with, and possibly the type of character encoding. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Possible encodings include Unicode character sets (such as UTF-8, UTF-16, and UTF-32) and some ISO encodings (such as ISO-10646-xxx or ISO-8859-xxx). The prologue can also include external declarations, the schema used to validate the document, namespace declarations, an associated XSL file, and some internal entity declarations. Refer to XML documentation or books for more information about these topics.

An XML document is well formed if it has a prologue, has a proper syntax (see the rules in the previous section), and has a tree of nodes with a single root. Most tools (including Internet Explorer) check whether a document is well formed when loading it.

NOTE *XML is more formal and precise than HTML. The W3C is working on an XHTML standard that will make HTML documents XML compliant, for better processing with XML tools. This implies many changes in a typical HTML document, such as avoiding attributes with no values, adding all the closing markers (as in* `</p>` *and* `</li>`*), adding the slash to stand-alone markers (as* `<hr/>` *and* `<br/>`*), proper nesting, and more. An HTML-to-XHTML converter called HTML Tidy is hosted by the W3C website at* `www.w3.org/People/Raggett/tidy/`.

Working with XML

To get acquainted with the format of XML, you can use one of the existing XML editors available on the market (including Delphi itself and Context, a programmer's editor written in Delphi). When you load an XML document into Internet Explorer, you'll see whether it is correct and, in this case, you'll see it within the browser in a tree-like structure. (At the time I'm writing this, other browsers have more limited XML support.)

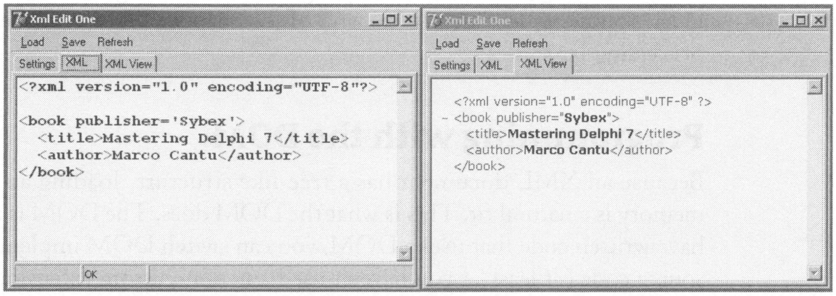
To speed up this type of operation, I've built the simplest XML editor I could come up with—basically a memo with XML syntax-checking and a browser attached to it. The XmlEditOne example has a PageControl with three pages. The first page, Settings, hosts a couple of components in which you can insert the path and the name of the file you want to work with. (The reason for not using a standard dialog will become clear when I show you an extension of the program.) The edit box hosting the complete filename is automatically updated with the path and filename, provided the AutoUpdate check box is selected.

The second page hosts a Memo control; the text of the XML file is loaded and saved by clicking the two toolbar buttons. As soon as you load the file, or each time you modify its text, its content is loaded into a DOM to let a parser check for its correctness (something that would be complex to do with your own code). To parse the code, I've used the XMLDocument component available in Delphi, which is basically a wrapper around a DOM available on the computer and indicated by its `DOMVendor` property. I'll discuss the use of this component in more detail in the next section. For the moment, suffice to say you can assign a string list to its `XML` property and activate it to let it parse the XML text and eventually report an error with an exception.

For this example, this behavior is far from good, because while typing the XML code you'll have temporarily incorrect XML. Still, I prefer not to ask the user to click a button to do the validation, but rather to let it run continuously. Because it is not possible to disable the parse exception raised by the XMLDocument component, I had to work at a lower level, extracting the `DOMPersist` property (referring to the persistency interface of the DOM) after extracting the `IXMLDocumentAccess` interface from the XMLDocument component (called `XmlDoc` in this code). You can also extract the `IDOMParseError` interface from the document component, to display any error message in the status bar:

```
procedure TFormXmlEdit.MemoXmlChange(Sender: TObject);
var
  eParse: IDOMParseError;
begin
  XmlDoc.Active := True;
  xmlBar.Panels[1].Text := 'OK';
  xmlBar.Panels[2].Text := '';
  (XmlDoc as IXMLDocumentAccess).DOMPersist.loadxml(MemoXml.Text);
  eParse := (XmlDoc.DOMDocument as IDOMParseError);
  if eParse.errorCode <> 0 then
    with eParse do
    begin
      xmlBar.Panels[1].Text := 'Error in: ' + IntToStr (Line) + '.' +
        IntToStr (LinePos);
      xmlBar.Panels[2].Text := SrcText + ': ' + Reason;
    end;
end;
```

You can see an example of the output of the program in Figure 22.1, alongside the XML tree view provided by the third page (for a correct document). The third page of the program is built using the WebBrowser component, which embeds Internet Explorer's ActiveX control. Unfortunately, there is no direct way to assign a string with the XML text to this control, so you'll have to save the file first and then move to its page to trigger the loading of the XML in the browser (after manually clicking the Refresh button at least once).



```
<?xml version="1.0" encoding="UTF-8"?>

<book publisher='Sybex'>
  <title>Mastering Delphi 7</title>
  <author>Marco Cantu</author>
</book>
```

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <book publisher="Sybex">
    <title>Mastering Delphi 7</title>
    <author>Marco Cantu</author>
  </book>
```

FIGURE 22.1 The XmlEditOne example allows you to enter XML text in a memo, indicating errors as you type, and shows the result in the embedded browser.

NOTE *I've used this code as a starting point to build a full-fledged XML editor called XmlTypist for a company I work with. It includes syntax highlighting, XSLT support, and a number of extra features. Refer to Appendix A, "Extra Delphi Tools by the Author" for the availability of this free XML editor.*

Managing XML Documents in Delphi

Now that you know the core elements of XML, we can begin discussing how to manage XML documents in Delphi programs (or in programs in general; some of the techniques discussed here go beyond the language used). There are two typical techniques for manipulating XML documents: using a Document Object Model (DOM) interface or using the Simple API for XML (SAX). The two approaches are quite different:

- The DOM loads the entire document into a hierarchical tree of nodes, allowing you to read them and manipulate them to change the document. For this reason, the DOM is suitable when you want to navigate the XML structure in memory and edit it, or for creating new documents from scratch.
- The SAX parses the document, firing an event for each element of the document without building a structure in memory. Once parsed by the SAX, the document is lost, but this operation is generally much faster than the construction of the DOM tree. Using the SAX is good for reading a document once—for example, if you're looking for a portion of its data.

There is a third classic way to manipulate (and specifically create) XML documents: string management. Creating a document by adding strings is the fastest operation, particularly if you can do a single pass (and don't need to modify nodes already generated). Even reading documents by means of string functions is very fast, but this process can become difficult for complex structures.

Besides these classic XML processing approaches, which are also available for other programming languages, Delphi provides two more techniques you should consider. The first is the definition of interfaces that map the document structure and are used to access the document instead of the generic DOM interface. As you'll see, this approach makes for faster coding and more robust applications. The second technique is the development of transformations that allow you to read a generic XML document into a ClientDataSet component or save the dataset into an XML file of a given structure (not the specific XML structure natively supported by the ClientDataSet or MyBase).

I won't try to fully assess which option is better suited for each type of document and manipulation, but I will highlight some of the advantages and disadvantages while discussing examples of each approach in the following sections. At the end of the chapter, I'll discuss the relative speed of techniques for processing large files.

Programming with the DOM

Because an XML document has a tree-like structure, loading an XML document into a tree in memory is a natural fit. This is what the DOM does. The DOM is a standard interface, so when you have written code that uses a DOM, you can switch DOM implementations without changing your source code (at least, if you haven't used any non-custom extensions).

In Delphi, you can install several DOM implementations, available as COM servers, and use their interfaces. One of the most commonly used DOM engines on Windows is the one provided by Microsoft as part of the MSXML SDK but also installed by Internet Explorer (and for this reason in all recent versions of Windows) and many other Microsoft applications. (With the full MSXML SDK also containing documentation and examples you don't get in other embedded installations of the same library.) Other DOM engines directly available in Delphi 7 include Apache Foundation's Xerces and the open-source OpenXML.

TIP *OpenXML is a native Object Pascal DOM available at* `www.philo.de/xml`. *Another native Delphi DOM is offered by TurboPower. These solutions offer two advantages: They don't require an external library for the program to execute, because the DOM component is compiled into your application; and they are cross-platform.*

Delphi embeds the DOM implementations into a wrapper component called XMLDocument. I used this component in the preceding example, but here I will examine its role in a more general way. The idea behind using this component instead of the DOM interface is that you remain more independent from the implementations and can work with some simplified methods, or helpers.

The DOM interface is complex to use. A *document* is a collection of nodes, each having a name, a text element, a collection of attributes, and a collection of child nodes. Each collection of nodes lets you access elements by position or search for them by name. Notice that the text within the tags of a node, if any, is rendered as a child of the node and listed in its collection of child nodes. The root node has some extra methods for creating new nodes, values, or attributes.

With Delphi's XMLDocument, you can work at two levels:

- At a lower level, you can use the `DOMDocument` property (of the `IDOMDocument` interface type) to access a standard W3C Document Object Model interface. The official DOM is defined in the xmldom unit and includes interfaces like `IDOMNode`, `IDOMNodeList`, `IDOMAttr`, `IDOMElement`, and `IDOMText`. With the official DOM interfaces, Delphi supports a lower-level but standard programming model. The DOM implementation is indicated by the XMLDocument component in the `DOMVendor` property.

- As a higher-level alternative, the XMLDocument component also implements the `IXMLDocument` interface. This is a custom DOM-like API defined by Borland in the XMLIntf unit and comprising interfaces like `IXMLNode`, `IXMLNodeList`, and `IXMLNodeCollection`. This Borland interface simplifies some of the DOM operations by replacing multiple method calls, which are repeated often in sequence, with a single property or method.

In the following examples (particularly the DomCreate demo), I'll use both approaches to give you a better idea of the practical differences between the two approaches.

An XML Document in a TreeView

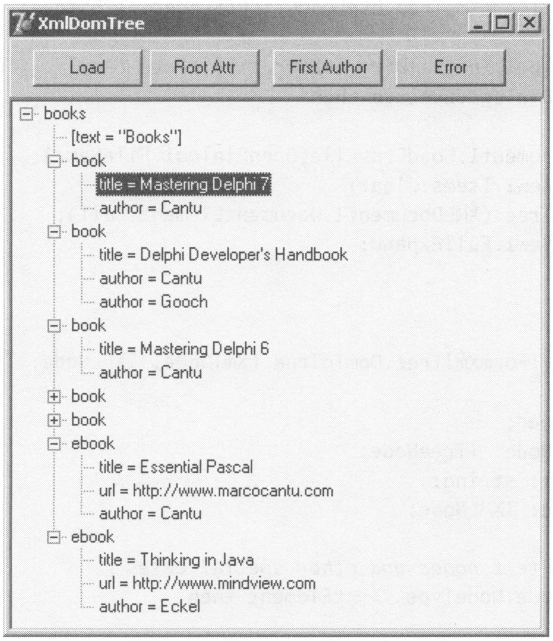
The starting point generally consists of loading a document from a file or creating it from a string, but you can also start with a new document. As a first example of using the DOM, I've built a program that can load an XML document into a DOM and show its structure in a TreeView control. I've also added to the XmlDomTree program a few buttons with sample code used to access to the elements of a sample file, as an example of accessing the DOM data. Loading the document is simple, but showing it in a tree requires a recursive function that navigates the nodes and subnodes. Here is the code for the two methods:

```
procedure TFormXmlTree.btnLoadClick(Sender: TObject);
begin
  OpenDialog1.InitialDir := ExtractFilePath (Application.ExeName);
  if OpenDialog1.Execute then
  begin
    XMLDocument1.LoadFromFile(OpenDialog1.FileName);
    Treeview1.Items.Clear;
    DomToTree (XMLDocument1.DocumentElement, nil);
    TreeView1.FullExpand;
  end;
end;

procedure TFormXmlTree.DomToTree (XmlNode: IXMLNode; TreeNode: TTreeNode);
var
  I: Integer;
  NewTreeNode: TTreeNode;
  NodeText: string;
  AttrNode: IXMLNode;
begin
  // skip text nodes and other special cases
  if XmlNode.NodeType <> ntElement then
    Exit;
  // add the node itself
  NodeText := XmlNode.NodeName;
  if XmlNode.IsTextElement then
    NodeText := NodeText + ' = ' + XmlNode.NodeValue;
  NewTreeNode := TreeView1.Items.AddChild(TreeNode, NodeText);
  // add attributes
  for I := 0 to xmlNode.AttributeNodes.Count - 1 do
  begin
    AttrNode := xmlNode.AttributeNodes.Nodes[I];
    TreeView1.Items.AddChild(NewTreeNode,
      '[' + AttrNode.NodeName + ' = "' + AttrNode.Text + '"]');
  end;
  // add each child node
  if XmlNode.HasChildNodes then
    for I := 0 to xmlNode.ChildNodes.Count - 1 do
      DomToTree (xmlNode.ChildNodes.Nodes [I], NewTreeNode);
end;
```

This code is interesting because it highlights some of the operations you can do with a DOM. First, each node has a `NodeType` property you can use to determine whether the node is an element, attribute, text node, or special entity (such as CDATA and others). Second, you cannot access the textual representation of the node (its `NodeValue`) unless it has a text element (notice that the text node will be skipped, as per the initial test). After displaying the name of the item, and then the text value if available, the program shows the content of each attribute directly and of each subnode by calling the `DomToTree` method recursively (see Figure 22.2).

FIGURE 22.2

The XmlDomTree example can open a generic XML document and show it inside a TreeView common control.



```
books
  [text = "Books"]
  book
    title = Mastering Delphi 7
    author = Cantu
  book
    title = Delphi Developer's Handbook
    author = Cantu
    author = Gooch
  book
    title = Mastering Delphi 6
    author = Cantu
  book
  book
  ebook
    title = Essential Pascal
    url = http://www.marcocantu.com
    author = Cantu
  ebook
    title = Thinking in Java
    url = http://www.mindview.com
    author = Eckel
```

Once you have loaded the sample document that accompanies the XmlDomTree program (shown in Listing 22.1) into the XMLDocument component, you can use the various methods to access generic nodes, as in the previous tree-building code, or fetch specific elements. For example, you can grab the value of the attribute *text* of the root node by writing:

```
XMLDocument1.DocumentElement.Attributes ['text']
```

Notice that if there is no attribute called *text,* the call will fail with a generic error message, "Invalid variant type conversion," which helps neither you nor the end user to understand what's wrong. If you need to access to the first attribute of the root without knowing its name, you can use the following code:

```
XMLDocument1.DocumentElement.AttributeNodes.Nodes[0].NodeValue
```

To access the nodes, you use a similar technique, possibly taking advantage of the `ChildValues` array. This is a Delphi extension to the DOM, which allows you to pass as parameter either the name of the element or its numeric position:

```
XMLDocument1.DocumentElement.ChildNodes.Nodes[1].ChildValues['author']
```

This code gets the (first) author of the second book. You cannot use the `ChildValues['book']` expression, because there are multiple nodes with the same name under the root node.

LISTING 22.1: THE SAMPLE XML DOCUMENT USED BY EXAMPLES IN THIS CHAPTER

```
<?xml version="1.0" encoding="UTF-8"?>
<books text="Books">
  <book>
    <title>Mastering Delphi 7</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi Developer's Handbook</title>
    <author>Cantu</author>
    <author>Gooch</author>
  </book>
  <book>
    <title>Delphi COM Programming</title>
    <author>Harmon</author>
  </book>
  <book>
    <title>Thinking in C++</title>
    <author>Eckel</author>
  </book>
  <ebook>
    <title>Essential Pascal</title>
    <url>http://www.marcocantu.com</url>
    <author>Cantu</author>
  </ebook>
  <ebook>
    <title>Thinking in Java</title>
    <url>http://www.mindview.com</url>
    <author>Eckel</author>
  </ebook>
</books>
```

Creating Documents Using the DOM

Although I mentioned earlier that you can create an XML document by chaining together strings, this technique is far from robust. Using a DOM to create a document ensures that the XML will be well formed. Also, if the DOM has a schema definition attached, you can validate the structure of the document while adding data to it.

To highlight different cases of document creation, I've built the DomCreate example. This program can create XML documents within the DOM, showing their text on a memo and optionally in a TreeView.

WARNING *The XMLDocument component uses the doAutoIndent option to improve the output of the XML text to the memo by formatting the XML in a slightly better way. You can choose the type of indentation by setting the* `NodeIndentStr` *property. To format generic XML text, you can also use the global* `FormatXMLData` *function using the default setting (two spaces) as indentation. Oddly, there doesn't seem a way to pass a different parameter to the function.*

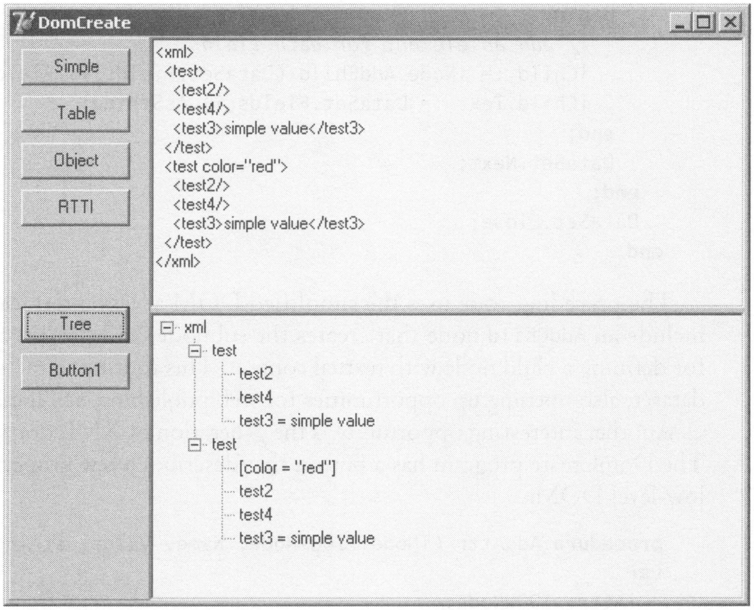
The Simple button on the form creates simple XML text using the low-level, official DOM interfaces. The program calls the document's `createElement` method for each node, adding them as children of other nodes:

```
procedure TForm1.btnSimpleClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot, iNode, iNode2, iChild, iAttribute: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('xml'));
  // node "test"
  iNode := iRoot.appendChild (iXml.createElement ('test'));
  iNode.appendChild (iXml.createElement ('test2'));
  iChild := iNode.appendChild (iXml.createElement ('test3'));
  iChild.appendChild (iXml.createTextNode('simple value'));
  iNode.insertBefore (iXml.createElement ('test4'), iChild);

  // node replication
  iNode2 := iNode.cloneNode (True);
  iRoot.appendChild (iNode2);

  // add an attribute
  iAttribute := iXml.createAttribute ('color');
  iAttribute.nodeValue := 'red';
  iNode2.attributes.setNamedItem (iAttribute);
  // show XML in memo
  Memo1.Lines.Text := FormatXMLData (XMLDoc.XML.Text);
end;
```

Notice that text nodes are added explicitly, attributes are created with a specific create call, and the code uses `cloneNode` to replicate an entire branch of the tree. Overall, the code is cumbersome to write, but after a while you may get used to this style. The effect of the program is shown (formatted in the memo and in the tree) in Figure 22.3.

FIGURE 22.3

The DomCreate example can generate various types of XML documents using a DOM.



```
<xml>
 <test>
  <test2/>
  <test4/>
  <test3>simple value</test3>
 </test>
 <test color="red">
  <test2/>
  <test4/>
  <test3>simple value</test3>
 </test>
</xml>
```

```
⊟ xml
   ⊟ test
       test2
       test4
       test3 = simple value
   ⊟ test
       [color = "red"]
       test2
       test4
       test3 = simple value
```

The second example of DOM creation relates to a dataset. I've added to the form a dbExpress dataset component (but any other dataset would do). I also added to a button the call to my custom `DataSetToDOM` procedure, like this:

```
DataSetToDOM ('customers', 'customer', XMLDoc, SQLDataSet1);
```

The `DataSetToDOM` procedure creates a root node with the text of the first parameter, grabs each record of the dataset, defines a node with the second parameter, and adds a subnode for each field of the record, all using extremely generic code:

```
procedure DataSetToDOM (RootName, RecordName: string; XMLDoc: TXMLDocument;
  DataSet: TDataSet);
var
  iNode, iChild: IXMLNode;
  i: Integer;
begin
  DataSet.Open;
  DataSet.First;
```

```
    // root
    XMLDoc.DocumentElement := XMLDoc.CreateNode (RootName);

    // add table data
    while not DataSet.EOF do
    begin
      // add a node for each record
      iNode := XMLDoc.DocumentElement.AddChild (RecordName);
      for I := 0 to DataSet.FieldCount - 1 do
      begin
        // add an element for each field
        iChild := iNode.AddChild (DataSet.Fields[i].FieldName);
        iChild.Text := DataSet.Fields[i].AsString;
      end;
      DataSet.Next;
    end;
    DataSet.Close;
  end;
```

The preceding code uses the simplified DOM access interfaces provided by Borland, which include an `AddChild` node that creates the subnode, and the direct access to the `Text` property for defining a child node with textual content. This routine extracts an XML representation of your dataset, also opening up opportunities for web publishing, as I'll discuss later in the section on XSL.

Another interesting opportunity is the generation of XML documents describing Delphi objects. The DomCreate program has a button that describes a few properties of an object, again using the low-level DOM:

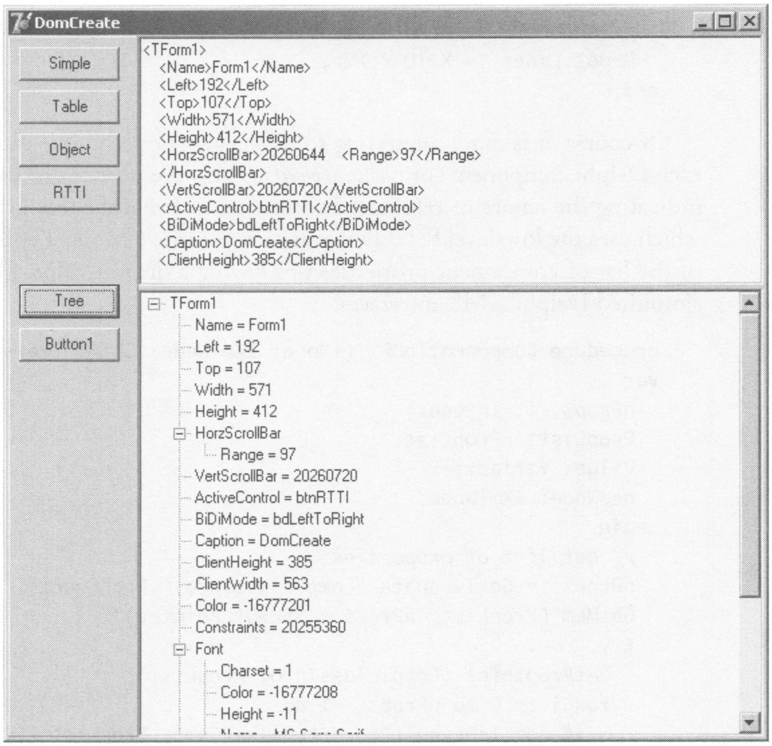
```
procedure AddAttr (iNode: IDOMNode; Name, Value: string);
var
  iAttr: IDOMNode;
begin
  iAttr := iNode.ownerDocument.createAttribute (name);
  iAttr.nodeValue := Value;
  iNode.attributes.setNamedItem (iAttr);
end;

procedure TForm1.btnObjectClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('Button1'));
  // a few properties as attributes (might also be nodes)
  AddAttr (iRoot, 'Name', Button1.Name);
  AddAttr (iRoot, 'Caption', Button1.Caption);
  AddAttr (iRoot, 'Font.Name', Button1.Font.Name);
  AddAttr (iRoot, 'Left', IntToStr (Button1.Left));
  AddAttr (iRoot, 'Hint', Button1.Hint);

  // show XML in memo
  Memo1.Lines := XmlDoc.XML;
end;
```

Of course, it is more interesting to have a generic technique capable of saving the properties of each Delphi component (or *persistent object*, to be more precise), recursing on persistent subobjects and indicating the names of referenced components. I've done this in the `ComponentToDOM` procedure, which uses the low-level RTTI information provided by the TypInfo unit, including the extraction of the list of component properties not having a default value. Once more, the program uses the simplified Delphi XML interfaces:

```
procedure ComponentToDOM (iNode: IXmlNode; Comp: TPersistent);
var
  nProps, i: Integer;
  PropList: PPropList;
  Value: Variant;
  newNode: IXmlNode;
begin
  // get list of properties
  nProps := GetTypeData (Comp.ClassInfo)^.PropCount;
  GetMem (PropList, nProps * SizeOf(Pointer));
  try
    GetPropInfos (Comp.ClassInfo, PropList);
    for i := 0 to nProps - 1 do
      if not IsDefaultPropertyValue(Comp, PropList [i], nil) then
      begin
        Value := GetPropValue (Comp, PropList [i].Name);
        NewNode := iNode.AddChild(PropList [i].Name);
        NewNode.Text := Value;
        if (PropList [i].PropType^.Kind = tkClass) and (Value <> 0) then
          if TObject (Integer(Value)) is TComponent then
            NewNode.Text := TComponent (Integer(Value)).Name
          else
            // TPersistent but not TComponent: recurse
            ComponentToDOM (newNode, TObject (Integer(Value)) as TPersistent);
      end;
  finally
    FreeMem (PropList);
  end;
end;
```

These two lines of code, in this case, trigger the creation of the XML document (shown in Figure 22.4):

```
XMLDoc.DocumentElement := XMLDoc.CreateNode(Self.ClassName);
ComponentToDOM (XMLDoc.DocumentElement, Self);
```



```
<TForm1>
  <Name>Form1</Name>
  <Left>192</Left>
  <Top>107</Top>
  <Width>571</Width>
  <Height>412</Height>
  <HorzScrollBar>20260644   <Range>97</Range>
  </HorzScrollBar>
  <VertScrollBar>20260720</VertScrollBar>
  <ActiveControl>btnRTTI</ActiveControl>
  <BiDiMode>bdLeftToRight</BiDiMode>
  <Caption>DomCreate</Caption>
  <ClientHeight>385</ClientHeight>
```

```
⊟ TForm1
    Name = Form1
    Left = 192
    Top = 107
    Width = 571
    Height = 412
  ⊟ HorzScrollBar
        Range = 97
    VertScrollBar = 20260720
    ActiveControl = btnRTTI
    BiDiMode = bdLeftToRight
    Caption = DomCreate
    ClientHeight = 385
    ClientWidth = 563
    Color = -16777201
    Constraints = 20255360
  ⊟ Font
        Charset = 1
        Color = -16777208
        Height = -11
```

FIGURE 22.4

The XML generated to describe the form of the DomCreate program. Notice (in the tree and in the memo text) that properties of class types are further expanded.

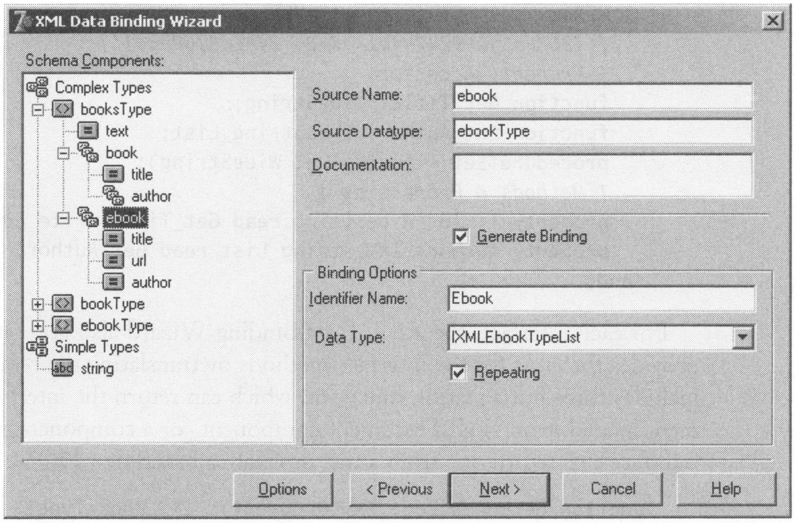
XML Data-Binding Interfaces

You have seen that working with the DOM to access or generate a document is tedious, because you must use positional information and not logical access to the data. Also, handling series of repeated nodes of different possible types (as shown in the XML sample in Listing 22.1, describing books) is far from simple. Moreover, using a DOM, you can create any well-formed document; but (unless you use a validating DOM) you can add any subnode to any node, coming up with almost useless documents, because no one else will be able to manage them.

To solve these issues, Borland added to Delphi an XML Data Binding Wizard, which can examine an XML document or a document definition (a schema, a Document Type Definition (DTD), or another type of definition) and generate a set of interfaces for manipulating the document. These interfaces are specific to the document and its structure and allow you to have more readable code, but they are certainly less generic as far as the types of documents you can handle (and this is more positive than it might sound at first).

You can activate the XML Data Binding Wizard by using the corresponding icon in the first page of the IDE's New Items dialog box or by double-clicking the XMLDocument component. (It is odd that the corresponding command is not in the shortcut menu of the component.)

After a first page in which you can select an input file, this wizard shows you the structure of the document graphically, as you can see in Figure 22.5 for the sample XML file from Listing 22.1. In this page, you can give a name to each entity of the generated interfaces, if you don't like the defaults suggested by the wizard. You can also change the rules used by the wizard to generate the names (an extended flexibility I'd like to have in other areas of the Delphi IDE). The final page gives you a preview of the generated interfaces and offers options for generating schemas and other definition files.



Source Name: ebook

Source Datatype: ebookType

Documentation:

☑ Generate Binding

Identifier Name: Ebook

Data Type: IXMLEbookTypeList

☑ Repeating

FIGURE 22.5
Delphi's XML Data Binding Wizard can examine the structure of a document or a schema (or another document definition) to create a set of interfaces for simplified and direct access to the DOM data.

For the sample XML file with the author names, the XML Data Binding Wizard generates an interface for the root node, two interfaces for the elements lists of two different types of nodes (books and e-books), and two interfaces for the elements of the two types. Here are a few excerpts of the generated code, available in the XmlIntfDefinition unit of the XmlInterface example:

```
type
  IXMLBooksType = interface(IXMLNode)
    ['{C9A9FB63-47ED-4F27-8ABA-E71F30BA7F11}']
    { Property Accessors }
    function Get_Text: WideString;
    function Get_Book: IXMLBookTypeList;
    function Get_Ebook: IXMLEbookTypeList;
    procedure Set_Text(Value: WideString);
    { Methods & Properties }
    property Text: WideString read Get_Text write Set_Text;
    property Book: IXMLBookTypeList read Get_Book;
    property Ebook: IXMLEbookTypeList read Get_Ebook;
  end;

  IXMLBookTypeList = interface(IXMLNodeCollection)
    ['{3449E8C4-3222-47B8-B2B2-38EE504790B6}']
    { Methods & Properties }
    function Add: IXMLBookType;
    function Insert(const Index: Integer): IXMLBookType;
    function Get_Item(Index: Integer): IXMLBookType;
    property Items[Index: Integer]: IXMLBookType read Get_Item; default;
  end;

  IXMLBookType = interface(IXMLNode)
    ['{26BF5C51-9247-4D1A-8584-24AE68969935}']
    { Property Accessors }
    function Get_Title: WideString;
    function Get_Author: IXMLString_List;
    procedure Set_Title(Value: WideString);
    { Methods & Properties }
    property Title: WideString read Get_Title write Set_Title;
    property Author: IXMLString_List read Get_Author;
  end;
```

For each interface, the XML Data Binding Wizard also generates an implementation class that provides the code for the interface methods by translating the requests into DOM calls. The unit includes three initialization functions, which can return the interface of the root node from a document loaded in an XMLDocument component (or a component providing a generic `IXMLDocument` interface), or return one from a file, or create a brand new DOM:

```
function Getbooks(Doc: IXMLDocument): IXMLBooksType;
function Loadbooks(const FileName: WideString): IXMLBooksType;
function Newbooks: IXMLBooksType;
```

After generating these interfaces using the wizard in the XmlInterface example, I've repeated XML document access code that's similar to the XmlDomTree example but is much simpler to write (and to read). For example, you can get the attribute of the root node by writing

```
procedure TForm1.btnAttrClick(Sender: TObject);
var
  Books: IXMLBooksType;
begin
  Books := Getbooks (XmlDocument1);
  ShowMessage (Books.Text);
end;
```

It is even simpler if you recall that while typing this code, Delphi's code insight can help by listing the available properties of each node, thanks to the fact that the parser can read in the interface definitions (although it cannot understand the format of a generic XML document). Accessing a node of one of the sublists is a matter of writing one of the following statements (possibly the second, with the default array property):

```
Books.Book.Items[1].Title  // full
Books.Book[1].Title        // further simplified
```

You can use similarly simplified code to generate new documents or add new elements, thanks to the customized `Add` method available in each list-based interface. Again, if you don't have a predefined structure for the XML document, as in the dataset-based and RTTI-based examples of the previous demonstration, you won't be able to use this approach.

VALIDATION AND SCHEMAS

The XML Data Binding Wizard can work from existing schemas or generate a schema for an XML document (and eventually save it in a file with the `.XDB` extension). An XML document describes some data, but to exchange this data among companies, it must stick to some agreed structure. A schema is a document definition against which a document can be checked for correctness, an operation usually indicated with the term *validation*.

The first—and still widespread—type of validation available for XML used *document type definitions (DTDs)*. These documents describe the structure of the XML but cannot define the possible content of each node. Also, DTDs are not XML document themselves, but use a different, awkward notation.

At the end of 2000, the W3C approved the first official draft of XML *schemas* (already available in an incompatible version called XML-Data within Microsoft's DOM). An XML schema is an XML document that can validate both the structure of the XML tree and the content of the node. A schema is based on the use and definition of simple and complex data types, similar to what happens in an OOP language.

A schema defines complex types, indicating for each the possible nodes, their optional sequence (`sequence`, `all`), the number of occurrences of each subnode (`minOccurs`, `maxOccurs`), and the data type of each specific element. Here is the schema defined by the XML Data Binding Wizard for the sample books file:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://www.borland.com/schemas/delphi/7.0/XMLDataBinding">
  <xs:element name="books" type="booksType"/>
  <xs:complexType name="booksType">
    <xs:annotation>
      <xs:appinfo xdb:docElement="books"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="book" type="bookType" maxOccurs="unbounded"/>
      <xs:element name="ebook" type="ebookType" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="text" type="xs:string"/>
  </xs:complexType>
```

```
    <xs:complexType name="bookType">
      <xs:annotation>
        <xs:appinfo xdb:repeated="True"/>
      </xs:annotation>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ebookType">
      <xs:annotation>
        <xs:appinfo xdb:repeated="True"/>
      </xs:annotation>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="url" type="xs:string"/>
        <xs:element name="author" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
```

Microsoft and Apache DOM engines have good support for schemas. Another tool I've used for validation is XML Schema Validator (XSV), an open-source attempt at a conformant schema-aware processor, which can be used either directly via the Web or after downloading a command-line executable (see the links to the current website of this tool in the W3C's XML Schema pages).

NOTE *The Delphi editor supports code completion for XML files via DTDs. Dropping a DTD file in Delphi's* `bin` *directory and referring to it with a* `DOCTYPE` *tag should enable this feature, which is not officially supported by Borland.*

Using the SAX API

The Simple API for XML (SAX) doesn't create a tree for the XML nodes, but parses the node—firing events for each node, attribute, value, and so on. Because it doesn't keep the document in memory, using the SAX allows you to manage much larger documents. Its approach is also useful for one-time examination of a document or retrieval of specific information. This is a list of the most important events fired by the SAX:

- `StartDocument` and `EndDocument` for the entire document
- `StartElement` and `EndElement` for each node
- `Characters` for the text within the nodes

It is common to use a stack to handle the current path within the nodes tree, and push and pop elements to and from the stack for every `StartElement` and `EndElement` event.

Delphi does not include specific support for the SAX interface, but you can import Microsoft's XML support (the MSXML library). In particular, for the SaxDemo1 example I've used version 2 of MSXML, because this version is widely available. I've generated a Pascal type library import unit from the type library, and the import unit is available within the source code of the program, but you must have that specific COM library registered on your computer to run the program successfully.

NOTE *Another example at the end of this chapter (LargeXml) demonstrates, among other things, the use of the SAX API, including the OpenXml engine.*

To use the SAX, you must install a SAX event handler within a SAX reader, and then load a file and parse it. I've used the SAX reader interface provided by MSXML for VB programmers. The official (C++) interface had a few errors in its type library that prevented Delphi from importing it properly. The main form of the SaxDemo1 example declares

```
sax: IVBSAXXMLReader;
```

In the `FormCreate` method, the `sax` variable is initialized with the COM object:

```
sax := CoSAXXMLReader.Create;
sax.ErrorHandler := TMySaxErrorHandler.Create;
```

The code also sets an error handler, which is a class implementing a specific interface (`IVBSAXErrorHandler`) with three methods that are called depending on the severity of the problem: `error`, `fatalError`, and `ignorableWarning`.

Simplifying the code a little, the SAX parser is activated by calling the `parseURL` method after assigning a content handler to it:

```
sax.ContentHandler := TMySaxHandler.Create;
sax.parseURL (filename)
```

So, the code ultimately resides in the `TMySaxHandler` class, which has the SAX events. Because I have multiple SAX content handlers in this example, I've written a base class with the core code and a few specialized versions for specific processing. Following is the code of the base class, which implements both the `IVBSAXContentHandler` interface and the `IDispatch` interface the `IVBSAXContentHandler` interface is based on:

```
type
  TMySaxHandler = class (TInterfacedObject, IVBSAXContentHandler)
  protected
    stack: TStringList;
  public
    constructor Create;
    destructor Destroy; override;
    // IDispatch
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
      HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer):
      HResult; stdcall;
    // IVBSAXContentHandler
    procedure Set_documentLocator(const Param1: IVBSAXLocator);
      virtual; safecall;
    procedure startDocument; virtual; safecall;
    procedure endDocument; virtual; safecall;
    procedure startPrefixMapping(var strPrefix: WideString;
      var strURI: WideString); virtual; safecall;
    procedure endPrefixMapping(var strPrefix: WideString); virtual; safecall;
    procedure startElement(var strNamespaceURI: WideString;
      var strLocalName: WideString; var strQName: WideString;
      const oAttributes: IVBSAXAttributes); virtual; safecall;
    procedure endElement(var strNamespaceURI: WideString;
      var strLocalName: WideString; var strQName: WideString);
      virtual; safecall;
    procedure characters(var strChars: WideString); virtual; safecall;
    procedure ignorableWhitespace(var strChars: WideString);
      virtual; safecall;
    procedure processingInstruction(var strTarget: WideString;
      var strData: WideString); virtual; safecall;
    procedure skippedEntity(var strName: WideString); virtual; safecall;
  end;
```

The most interesting portion, of course, is the final list of SAX events. All this base class does is emit information to a log when the parser starts (`startDocument`) and finishes (`endDocument`) and keep track of the current node and its parent nodes with a stack:

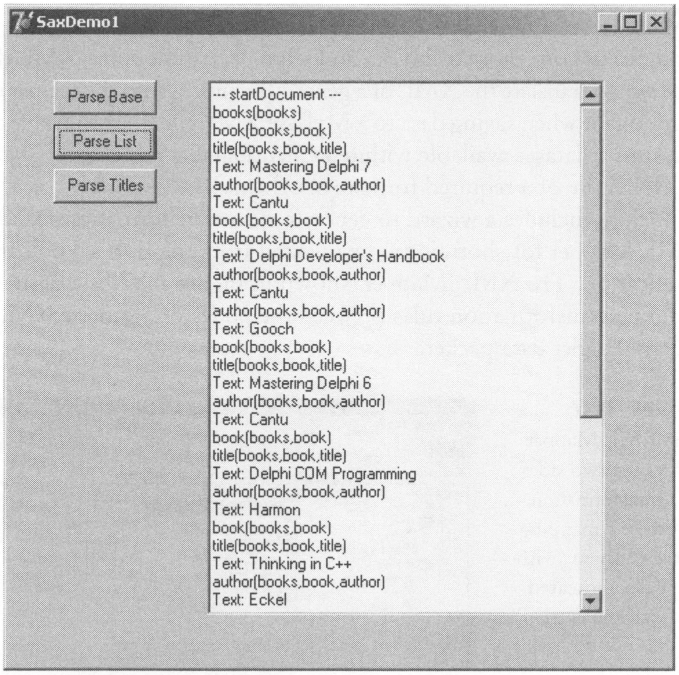
```
// TMySaxHandler.startElement
stack.Add (strLocalName);
// TMySaxHandler.endElement
stack.Delete (stack.Count - 1);
```

An implementation is provided by the `TMySimpleSaxHandler` class, which overrides the `startElement` event triggered for any new node to output the current position in the tree with the following statement:

```
Log.Add (strLocalName + '(' + stack.CommaText + ')');
```

The second method of the class is the `characters` event, which is triggered when a node value (or a test node) is encountered and outputs its content (as you can see in Figure 22.6):

```
procedure TMySimpleSaxHandler.characters(var strChars: WideString);
var
  str: WideString;
begin
  inherited;
  str := RemoveWhites (strChars);
  if (str <> '') then
    Log.Add ('Text: ' + str);
end;
```



```
--- startDocument ---
books(books)
book(books,book)
title(books,book,title)
Text: Mastering Delphi 7
author(books,book,author)
Text: Cantu
book(books,book)
title(books,book,title)
Text: Delphi Developer's Handbook
author(books,book,author)
Text: Cantu
author(books,book,author)
Text: Gooch
book(books,book)
title(books,book,title)
Text: Mastering Delphi 6
author(books,book,author)
Text: Cantu
book(books,book)
title(books,book,title)
Text: Delphi COM Programming
author(books,book,author)
Text: Harmon
book(books,book)
title(books,book,title)
Text: Thinking in C++
author(books,book,author)
Text: Eckel
```

FIGURE 22.6
The log produced by reading an XML document with the SAX in the SaxDemo1 example

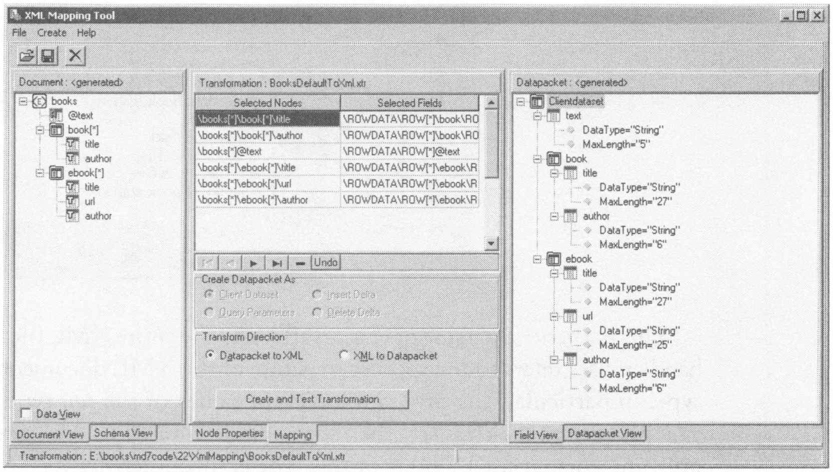
This is a generic parsing operation affecting the entire XML file. The second derived SAX content handler class refers to the specific structure of the XML document, extracting only nodes of a given type. In particular, the program looks for nodes of the *title* type. When a node has this type (in `startElement`), the class sets the `isbook` Boolean variable. The text value of the node is considered only immediately after a node of this type is encountered:

```
procedure TMyBooksListSaxHandler.startElement(var strNamespaceURI,
  strLocalName, strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
  inherited;
  isbook := (strLocalName = 'title');
end;

procedure TMyBooksListSaxHandler.characters(var strChars: WideString);
var
  str: string;
begin
  inherited;
  if isbook then
  begin
    str := RemoveWhites (strChars);
    if (str <> '') then
      Log.Add (stack.CommaText + ': ' + str);
  end;
end;
```

Mapping XML with Transformations

You can use one more technique in Delphi to handle some XML documents: You can create a *transformation* to translate the XML of a generic document into the format used natively by the ClientDataSet component when saving data to a MyBase XML file. In the reverse direction, another transformation can turn a dataset available within a ClientDataSet (through a DataSetProvider component) into an XML file of a required format (or schema).

Delphi includes a wizard to generate such transformations. Called the XML Mapping Tool, or XML Mapper for short, it can be invoked from the IDE's Tools menu or executed as a stand-alone application. The XML Mapper, shown in Figure 22.7, is a design-time helper that assists you in defining transformation rules between the nodes of a generic XML document and fields of the ClientDataSet data packet.


XML Mapping Tool
File Create Help
Document : <generated>
books
@text
book[*]
title
author
ebook[*]
title
url
author
Datapacket : <generated>
Clientdataset
text
DataType="String"
MaxLength="5"
book
title
DataType="String"
MaxLength="27"
author
DataType="String"
MaxLength="6"
ebook
title
DataType="String"
MaxLength="27"
url
DataType="String"
MaxLength="25"
author
DataType="String"
MaxLength="6"
Undo
Create and Test Transformation
Document View
Schema View
Node Properties
Mapping
Field View
Datapacket View

Transformation : BooksDefaultToXml.xtr

Selected Nodes	Selected Fields
\books[*]\book[*]\title	\ROWDATA\ROW[*]\book\RO
\books[*]\book[*]\author	\ROWDATA\ROW[*]\book\RO
\books[*]@text	\ROWDATA\ROW[*]@text
\books[*]\ebook[*]\title	\ROWDATA\ROW[*]\ebook\R
\books[*]\ebook[*]\url	\ROWDATA\ROW[*]\ebook\R
\books[*]\ebook[*]\author	\ROWDATA\ROW[*]\ebook\R

Create Datapacket As
● Client Dataset ○ Insert Delta
○ Query Parameters ○ Delete Delta

Transform Direction
● Datapacket to XML ○ XML to Datapacket

☐ Data View

Transformation : E:\books\md7code\22\XmlMapping\BooksDefaultToXml.xtr

FIGURE 22.7
The XML Mapper shows the two sides of a transformation to define a mapping between them (with the rules indicated in the central portion).

The XML Mapper window has three areas:

- On the left is the XML document section, which displays information about the structure of the XML document (and eventually its data, if the related check box is active) in the Document View or an XML schema in the Schema View, depending on the selected tab.
- On the right is the data packet section, which displays information about the metadata in the data packet, either in the Field View (indicating the dataset structure) or in the Datapacket View (reporting the XML structure). The XML Mapper can also open files in the native ClientDataSet format.
- The central portion is used by the mapping section. It contains two pages: Mapping, where you can see the correspondence between selected elements of the two sides that will be part of the mapping; and Node Properties, where you can modify the data types and other details of each possible mapping.

The Mapping page of the central pane also hosts the shortcut menu used to generate the transformation. The other panes and views have specific shortcut menus you can use to perform the various actions (besides the few commands in the main menu).

You can use XML Mapper to map an existing schema (or extract it from a document) to a new data packet, an existing data packet to a new schema or document, or an existing data packet into an existing XML document (if a match is reasonable). In addition to converting the data of an XML file into a data packet, you can also convert to a delta packet of the ClientDataSet. This technique is useful for merging a document to an existing table, as if a user had inserted the modified table records. In particular, you can transform an XML document into a delta packet for records to be modified, deleted, or inserted.

The result of using the XML Mapper is one or more transformation files, each representing a one-way conversion (so you need at least two transformation files to convert data back and forth). These transformation files are then used at design time and at run time by the XMLTransform, XMLTransformProvider, and XMLTransformClient components.

As an example, I opened the books XML document, which has a structure that doesn't easily match a table, because it includes two lists of values of different types (I've skipped easier examples in which the XML has a plain rectangular structure). After opening the `Sample.XML` file in the XML Document section, I used its shortcut menu to select all of its elements (Select All) and to create the data packet (Create Datapacket From XML). This command automatically fills the right pane with the data packet and the central portion with the proposed transformation. You can also view its effect in a sample program by clicking the Create And Test Transformation button. Doing so opens a generic application that can load a document into the dataset using the transformation you've just created.

In this case, the XML Mapper generates a table with two dataset fields: one for each possible list of subelements. This was the only possible standard solution, because the two sublists have different structures, and it is the only solution that allows you to edit the data in a DBGrid attached to the ClientDataSet and save it back to a complete XML file, as demonstrated by the XmlMapping example. This program is basically a Windows-based editor for a complex XML document.

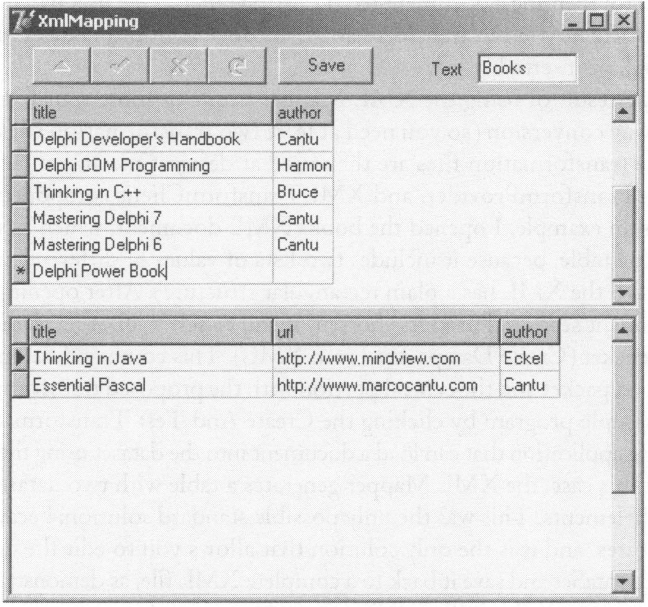
The example uses a TransformProvider component with two transformation files attached to read in an XML document and make it available to a ClientDataSet. As the name suggests, this component is a dataset provider. To build the user interface, I didn't connect the ClientDataSet directly to a grid, because it has a single record with a text field plus two detailed datasets. For this reason, I added to the program two more ClientDataSet components attached to the dataset fields and connected to two DBGrid controls. This explanation is easier to understand by looking at the definition of the non-visual components from DFM source code in the following excerpt and at its output in Figure 22.8.

```
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'BooksDefault.xtr'
  TransformWrite.TransformationFile = 'BooksDefaultToXml.xtr'
  XMLDataFile = 'Sample.xml'
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'XMLTransformProvider1'
  object ClientDataSet1text: TStringField
  object ClientDataSet1book: TDataSetField
  object ClientDataSet1ebook: TDataSetField
end
object ClientDataSet2: TClientDataSet
  DataSetField = ClientDataSet1book
end
```

```
object ClientDataSet3: TClientDataSet
  DataSetField = ClientDataSet1ebook
end
```

FIGURE 22.8

The XmlMapping example uses a TransformProvider component to make a complex XML document available for editing within multiple ClientDataSet components.



Text: Books

	title	author
	Delphi Developer's Handbook	Cantu
	Delphi COM Programming	Harmon
	Thinking in C++	Bruce
	Mastering Delphi 7	Cantu
	Mastering Delphi 6	Cantu
*	Delphi Power Book	

	title	url	author
▶	Thinking in Java	http://www.mindview.com	Eckel
	Essential Pascal	http://www.marcocantu.com	Cantu

This program allows you to edit the data of the various sublists of nodes within the grids, modifying them and also adding or deleting records. As you apply the changes to the dataset (clicking the Save button, which calls `ApplyUdpates`), the transform provider saves an updated version of the file to disk.

As an alternative approach, you can also create transformations that map only portions of the XML document into a dataset. As an example, see the `BooksOnly.xtr` file in the folder of the XmlMapping example. The modified XML document you'll generate will have a different structure and content from the original, including only the portion you've selected. So, it can be useful for viewing the data, but not for editing it.

NOTE *It is not surprising that the transformation files are themselves XML documents, as you can see by opening one in the editor. This XML document uses a custom format.*

At the opposite side, you can see how a transformation can be used to take a database table or the result of a query and produce an XML file with a more readable format than that provided by default by the ClientDataSet persistence mechanism. To build the MapTable example, I placed a dbExpress SimpleDataSet component on a form and attached a DataSetProvider to it and a ClientDataSet to the provider. After opening the table and the client dataset, I saved its content to an XML file.

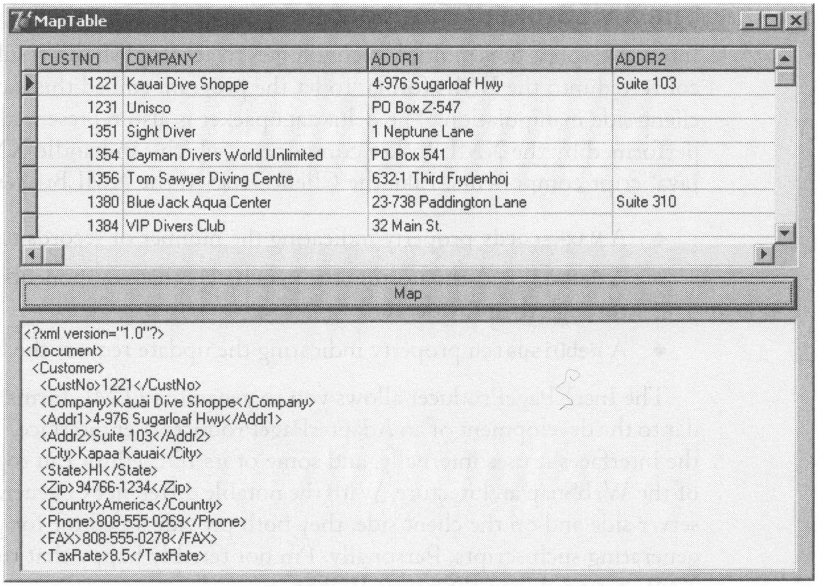
At that point, I opened the XML Mapper, loaded the data packet file into it, selected all the data packet nodes (with the Select All command from the shortcut menu) and invoked the Create XML

From Datapacket command. In the following dialog box, I accepted the default name mappings for fields and only changed the suggested name for record nodes (ROW) into something more readable (Customer). If you now test the transformation, the XML Mapper will display the contents of the resulting XML document in a custom tree view.

After saving the transformation file, I was ready to resume developing the program, removing the ClientDataSet and adding a DataSource and a DBGrid (as a user might edit in on an attached DBGrid before transforming it), and an XMLTransformClient component. This component has the transformation file connected to it, but not an XML file. Instead, it refers to the data through the provider. Clicking the button shows the XML document within a memo (after formatting it) instead of saving it to a file, something you can do by calling the `GetDataAsXml` method (even if the Help file is far from clear about the use of this method):

```
procedure TForm1.btnMapClick(Sender: TObject);
begin
  Memo1.Lines.Text := FormatXmlData(XMLTransformClient1.GetDataAsXml(''));
end;
```

This is the only code for the program visible at run time in Figure 22.9; you can see the original dataset in the DBGrid and the resulting XML document in the memo control below the grid. The application has much simpler code than the DomCreate example I used to generate a similar XML document, but it requires the design-time definition of the transformation. The DomCreate example could work on any dataset at run time without any connection to a specific table, because it has rather generic code. In theory, it is possible to produce similar dynamic mappings by using the events of the generic XMLTransform component, but I find it easier to use the DOM-based approach discussed earlier. Notice also that the `FormatXmlData` call produces nicer output but slows down the program, because it involves loading the XML into a DOM.



CUSTNO	COMPANY	ADDR1	ADDR2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	

```
<?xml version="1.0"?>
<Document>
 <Customer>
  <CustNo>1221</CustNo>
  <Company>Kauai Dive Shoppe</Company>
  <Addr1>4-976 Sugarloaf Hwy</Addr1>
  <Addr2>Suite 103</Addr2>
  <City>Kapaa Kauai</City>
  <State>HI</State>
  <Zip>94766-1234</Zip>
  <Country>America</Country>
  <Phone>808-555-0269</Phone>
  <FAX>808-555-0278</FAX>
  <TaxRate>8.5</TaxRate>
```

FIGURE 22.9
The MapTable example generates an XML document from a database table using a custom transformation file.

XML and Internet Express

Once you have defined the structure of an XML document, you might want to let users see and edit the data in a Windows application or over the Web. This second case is interesting because Delphi provides specific support for it. Delphi 5 introduced an architecture called Internet Express, which is now available as part of the WebSnap platform. WebSnap also offers support for XSL, which I'll discuss later in this chapter.

In Chapter 16, "Multitier DataSnap Applications," I discussed the development of DataSnap applications. Internet Express provides a client component called XMLBroker for this architecture, which can be used in place of a client dataset to retrieve data from a middle-tier DataSnap program and make it available to a specific type of page producer called InetXPageProducer. You can use these components in a standard WebBroker application or in a WebSnap program. The idea behind Internet Express is that you write a web server extension (as discussed in Chapter 20, "Web Programming with WebBroker and WebSnap"), which in turn produces web pages hooked to your DataSnap server. Your custom application acts as a DataSnap client and produces pages for a browser client. Internet Express offers the services required to build this custom application easily.

I know this sounds confusing, but Internet Express is a four-tier architecture: SQL server, application server (the DataSnap server), web server with a custom application, and web browser. Of course, you can place the database access components within the same application handling the HTTP request and generating the resulting HTML, as in a client/server solution. You can even access a local database or an XML file, in a two-tier structure (the server program and the browser).

In other words, Internet Express is a technology for building clients based on a browser, which lets you send the entire dataset to the client computer along with the HTML and some JavaScript for manipulating the XML and showing it into the user interface defined by the HTML. The JavaScript enables the browser to show the data and manipulate it.

The XMLBroker Component

Internet Express uses multiple technologies to accomplish this result. The DataSnap data packets are converted into the XML format to let the program embed this data into the HTML page for web client-side manipulation. The delta data packet is also represented in XML. These operations are performed by the XMLBroker component, which can handle XML and provide data to the new JavaScript components. Like the ClientDataSet, the XMLBroker has the following:

- A `MaxRecords` property indicating the number of records to add to a single page
- A `Params` property hosting the parameters that components will forward to the remote query through the provider
- A `WebDispatch` property indicating the update request the broker responds to

The InetXPageProducer allows you to generate HTML forms from datasets in a visual way similar to the development of an AdapterPageProducer user interface. The Internet Express architecture, the interfaces it uses internally, and some of its IDE editor can together be considered the parent of the WebSnap architecture. With the notable difference of generating scripts to be executed on the server side and on the client side, they both provide an editor for placing visual components and generating such scripts. Personally, I'm not terribly happy that the older Internet Express is more XML-oriented than the newer WebSnap.

TIP Another common feature of the InetXPageProducer and the AdapterPageProducer is the support for Cascading Style Sheets (CSS). These components have the `Style` *and* `StylesFile` *properties for defining the CSS, and each visual element has a* `StyleRule` *property you can use to select the style name.*

JavaScript Support

To make the editing operations on the client side powerful, the InetXPageProducer uses special JavaScript components and code. Delphi embeds a large JavaScript library, which the browser must download. This process might seem a nuisance, but it is the only way the browser interface (which is based on dynamic HTML) can be rich enough to support field constraints and other business rules with the browser. This functionality is impossible with plain HTML. The JavaScript files provided by Borland, which you should make available on the website hosting the application, are the following:

File	Description
`Xmldom.js`	DOM-compatible XML parser (for browsers lacking native XML DOM support)
`Xmldb.js`	JavaScript classes for the HTML controls
`Xmldisp.js`	JavaScript classes for binding XML data with the HTML controls
`Xmlerrdisp.js`	Classes for reconciling errors
`XmlShow.js`	JavaScript functions to display data and delta packets (for debugging purposes)

HTML pages generated by Internet Express usually include references to these JavaScript files, such as:

```
<script language=Javascript type="text/javascript"
  src="IncludePathURL/xmldb.js"></script>
```

You can customize the JavaScript by adding code directly to the HTML pages or by creating new Delphi components written to fit with the Internet Express architecture that emits JavaScript code (possibly along with HTML). As an example, the sample `TPromptQueryButton` class of INetXCustom generates the following HTML and JavaScript code:

```
<script language=javascript type="text/javascript">
  function PromptSetField(input, msg) {
    var v = prompt(msg);
    if (v == null || v == "")
      return false;
    input.value = v
    return true;
  }
  var QueryForm3 = document.forms['QueryForm3'];
</script>
<input type=button value="Prompt..."
  onclick="if (PromptSetField(PromptResult, 'Enter some text\n'))
    QueryForm3.submit();">
```

TIP If you plan to use Internet Express, look at the INetXCustom extra demo components, available in the `\Demos\Midas\InternetExpress\INetXCustom` *folder. Follow the detailed instructions in the* `readme.txt` *file to install these components, which are provided by Borland with no support but allow you to add many more features to your Internet Express applications with little extra effort.*

To deploy this architecture you don't need anything special on the client side, because any browser up to the HTML 4 standard can be used, on any operating system. The web server, however, must be a Win32 server (this technology is not available in Kylix), and you must deploy the DataSnap libraries on it.

Building an Example

To better understand what I'm talking about, and as a way to cover more technical details, let's try a demo called IeFirst. To avoid configuration issues, this is a CGI application accessing a dataset directly—in this case, a local table retrieved via a ClientDataSet component. Later, I'll show you how to turn an existing DataSnap Windows client into a browser-based interface. To build IeFirst, I created a new CGI application and added to its data module a ClientDataSet component hooked to a local .CDS file and a DataSetProvider component connected with the dataset. The next step is to add an XMLBroker component and connect it to the provider:

```
object ClientDataSet1: TClientDataSet
  FileName = 'C:\Program Files\Common Files\Borland
    Shared\Data\employee.cds'
end
object DataSetProvider1: TDataSetProvider
  DataSet = ClientDataSet1
end
object XMLBroker1: TXMLBroker
  ProviderName = 'DataSetProvider1'
  WebDispatch.MethodType = mtAny
  WebDispatch.PathInfo = 'XMLBroker1'
  ReconcileProducer = PageProducer1
  OnGetResponse = XMLBroker1GetResponse
end
```

The `ReconcileProducer` property is required to show a proper error message in case of an update conflict. As you'll see later, one of the Delphi demos includes some custom code, but in this example I've connected a traditional PageProducer component with a generic HTML error message. After setting up the XML broker, you can add an InetXPageProducer to the web data module. This component has a standard HTML skeleton; I've customized it to add a title, without touching the special tags:

```
<HTML><HEAD>
  <title>IeFirst</title>
</HEAD><BODY>
  <h1>Internet Express First Demo (IeFirst.exe)</h1>
  <#INCLUDES><#STYLES><#WARNINGS><#FORMS><#SCRIPT>
</BODY>
```

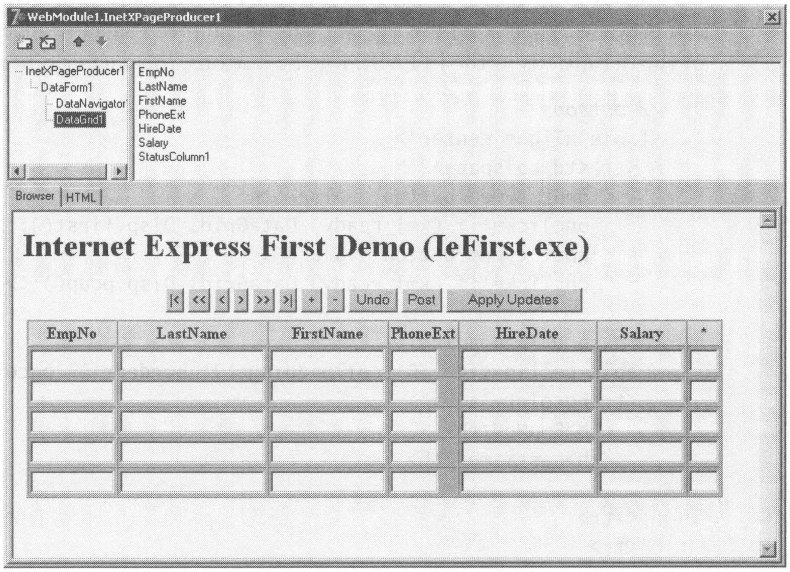
The special tags are automatically expanded using the JavaScript files of the directory specified by the `IncludePathURL` property. You *must* set this property to refer to the web server directory where these files reside. You can find them in the `\Delphi7\Source\WebMidas` directory. The five tags have the following effect:

Tag	Effect
`<#INCLUDES>`	Generates the instructions to include the JavaScript libraries
`<#STYLES>`	Adds the embedded style sheet definition
`<#WARNINGS>`	Used at design time to show errors in the InetXPageProducer editor
`<#FORMS>`	Generates the HTML code produced by the components of the web page
`<#SCRIPT>`	Adds a JavaScript block used to start the client-side script

NOTE *The InetXPageProducer component also handles a few more internal tags.* `<#BODYELEMENTS>` *corresponds to all five tags of the predefined template.* `<#COMPONENT Name=WebComponentName>` *is part of the generated HTML code used to declare the components generated visually.* `<#DATAPACKET XMLBroker=BrokerName>` *is replaced with the XML of the data packet.*

To customize the resulting HTML of the InetXPageProducer, you can use its editor, which again is similar to the one for WebSnap server-side scripting. Double-click the InetXPageProducer component, and Delphi opens a window like the one shown in Figure 22.10 (with the example's final settings). In this editor, you can create complex structures starting with a query form, data form, or generic layout group. In the example data form, I added a DataGrid and a DataNavigator component without customizing them any further (an operation you do by adding child buttons, columns, and other objects, which fully replace the defaults).

FIGURE 22.10
The InetXPageProducer editor allows you to build complex HTML forms visually, similarly to the AdapterPageProducer.



WebModule1.InetXPageProducer1
InetXPageProducer1
DataForm1
DataNavigator
DataGrid1
EmpNo
LastName
FirstName
PhoneExt
HireDate
Salary
StatusColumn1
Browser
HTML

Internet Express First Demo (IeFirst.exe)

|<
<<
<
>
>>
>|
+
-
Undo
Post
Apply Updates
EmpNo
LastName
FirstName
PhoneExt
HireDate
Salary
*

The DFM code for the InetXPageProducer and its internal components in my example is as follows. You can see the core settings plus some limited graphical customizations:

```
object InetXPageProducer1: TInetXPageProducer
  IncludePathURL = '/jssource/'
  HTMLDoc.Strings = (...)
  object DataForm1: TDataForm
    object DataNavigator1: TDataNavigator
      XMLComponent = DataGrid1
      Custom = 'align="center"'
    end
    object DataGrid1: TDataGrid
      XMLBroker = XMLBroker1
      DisplayRows = 5
      TableAttributes.BgColor = 'Silver'
      TableAttributes.CellSpacing = 0
      TableAttributes.CellPadding = 2
      HeadingAttributes.BgColor = 'Aqua'
      object EmpNo: TTextColumn...
      object LastName: TTextColumn...
      object FirstName: TTextColumn...
      object PhoneExt: TTextColumn...
      object HireDate: TTextColumn...
      object Salary: TTextColumn...
      object StatusColumn1: TStatusColumn...
    end
  end
end
```

The value of these components is in the HTML (and JavaScript) code they generate, which you can preview by selecting the HTML tab of the InetXPageProducer editor. Here are a few pieces of the definitions in the HTML, for the buttons, the data grid heading, and one of the grid's cells:

```
// buttons
<table align="center">
  <tr><td colspan="2">
    <input type="button" value="|<"
      onclick='if (xml_ready) DataGrid1_Disp.first();'>
    <input type="button" value="<<"
      onclick='if (xml_ready) DataGrid1_Disp.pgup();'>
...
// data grid heading
<table cellspacing="0" cellpadding="2" border="1" bgcolor="silver">
  <tr bgcolor="aqua">
    <th>EmpNo</th>
    <th>LastName</th>
  ...
  </tr>
  <tr>
```

```
// a data cell
<td><div>
  <input type="text" name="EmpNo" size="10"
    onfocus='if(xml_ready)DataGrid1_Disp.xfocus(this);'
    onkeydown='if(xml_ready)DataGrid1_Disp.keys(this);'>
</div></td>...
```

When the HTML generator is set up, you can go back to the web data module, add an action to it, and connect the action with the InetXPageProducer via the `Producer` property. This should be enough to make the program work through a browser, as you can see in Figure 22.11.

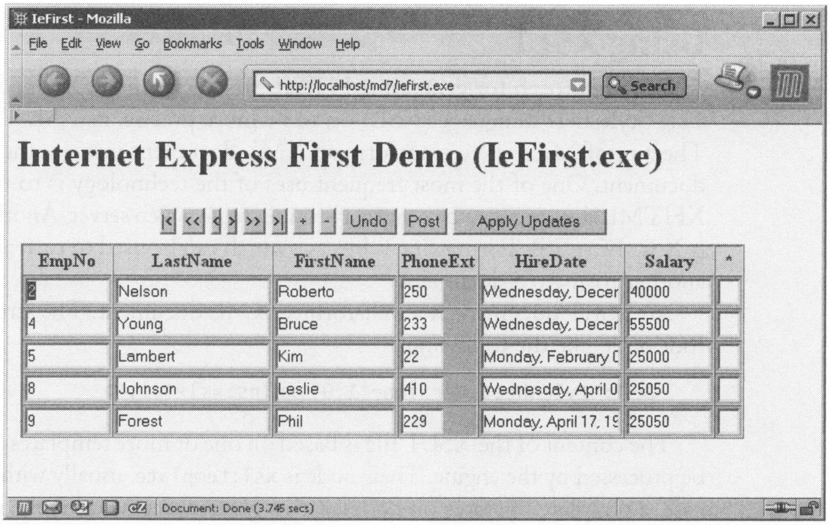


FIGURE 22.11

The IeFirst application sends the browser some HTML components, an XML document, and the JavaScript code to show the data in the visual components.

IeFirst - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost/md7/iefirst.exe
Search
Internet Express First Demo (IeFirst.exe)
|< << < > >> >| + - Undo Post Apply Updates
Document: Done (3.745 secs)

EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary	*
2	Nelson	Roberto	250	Wednesday, Decer	40000	
4	Young	Bruce	233	Wednesday, Decer	55500	
5	Lambert	Kim	22	Monday, February (	25000	
8	Johnson	Leslie	410	Wednesday, April 0	25050	
9	Forest	Phil	229	Monday, April 17, 1	25050	

If you look at the HTML file received by the browser, you'll find the table mentioned in the preceding definition, some JavaScript code, and the database data in the data packet XML format. This data is assembled by the XML broker and passed to the producer component to be embedded in the HTML file. The number of records sent to the client depends on the XMLBroker, not on the number of lines in the grid. Once the XML data is sent to the browser, you can use the navigator component's buttons to move around in the data without requiring further access to the server to fetch more. This is different from the paging behavior of WebSnap. One is not necessarily better than the other; it depends on the specific application you are building.

At the same time, the JavaScript classes in the system allow the user to type in new data, following the rules imposed by the JavaScript code hooked to dynamic HTML events. By default, the grid has an extra asterisk column indicating which records have been modified. The update data is collected in an XML data packet in the browser and sent back to the server when the user clicks the Apply Updates button. At this point, the browser activates the action specified by the `WebDispatch.PathInfo` property of the XMLBroker. There is no need to export this action from the web data module, because this operation is automatic (although you can disable it by setting `WebDispatch.Enable` to False).

The XMLBroker applies the changes to the server, returning the content of the provider connected to the `ReconcileProvider` property (or raising an exception if this property is not defined). When everything works fine, the XMLBroker redirects the control to the main page that contains the data. However, I've experienced some problems with this technique, so the IeFirst example handles the `OnGetResponse`, indicating this is an update view:

```
procedure TWebModule1.XMLBroker1GetResponse(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<h1>Updated</h1><p>' + InetXPageProducer1.Content;
  Handled := True;
end;
```

Using XSLT

Another approach for generating HTML starting from an XML document is the use of the Extensible Stylesheet Language (XSL) or, to be more precise, its XSL Transformations (XSLT) subset. The aim of XSLT is to transform an XML document into another document, generally an XML document. One of the most frequent uses of the technology is to turn an XML document into an XHTML document to be sent to a browser from a web server. Another interesting related technology is XSL-FO (XSL Formatting Objects), which can be used to turn an XML document into a PDF or another formatted document.

An XSLT document is a well-formed XML document. The structure of an XSLT file requires a root node like the following:

```
<xsl:stylesheet version="1.0" xmlns:xsl="...">
```

The content of the XSLT file is based on one or more templates (or rules or functions), which will be processed by the engine. Their node is `xsl:template`, usually with a `match` attribute. In the simplest case, a template operates on nodes with a given name; you invoke the template by passing to it one or more nodes with an XPath expression:

```
xsl:apply-templates select="node_name"
```

The starting point for this operation is a template that processes the root node, which can be the only template of the XSLT file. Within templates, you can find any of the other commands, such as the extraction of a value from an XML document (`xsl:value-of select`), looping statements (`xsl:for-each`), conditional expressions (`xsl:if`, `xsl:choose`), sorting requests (`xsl:sort`), and numbering requests (`xsl:number`), just to mention a few common XSLT commands.

Using XPath

XSLT uses other XML technologies, notably XPath to identify portions of documents. XPath defines a set of rules to locate one or more nodes within a document. The rules are based on a path-line structure of the node within the XML tree, so that `/books/book` identifies any *book* node under the *books* document root. XPath uses special symbols to identify nodes:

- A star (*) stands for any node; for example, `book/*` indicates any subnode under a *book* node.
- A dot (.) stands for the current node.
- The pipe symbol (|) indicates alternatives, as in `book|ebook`.

- A double slash (`//`) stands for any path, so that `//title` indicates all the title nodes, whatever their parent nodes, and `books//author` indicates any `author` node under a `books` node regardless of the nodes in between.
- The at sign (`@`) indicates an attribute instead of a node, as in the hypothetical `author/@lastname`.
- Square brackets can be used to choose only nodes or attributes having a given value. For example, to select all authors with a given first name attribute, you can use `author[@name="marco"]`.

There are many other cases, but this short introduction to the rules of XPath should get you started and help you understand the following examples. An XSLT document is an XML document that works on the structure of a source XML document and generates in output another XML document, such as an XHTML document you can view in a web browser.

NOTE *Commonly used XSLT processors include MS-XML, Xalan from the Apache XML project (`xml.apache.org`), and the Java-based Xt from James Clarke. From Delphi, you can also use the XSLT engine, included in TurboPower's XML Partner Pro (`www.turbopower.com`).*

XSLT in Practice

Let's discuss a couple of examples. As a starting point, you should study XSL by itself, and then focus on its activation from within a Delphi application.

For your initial tests, you can connect an XSL file directly to an XML file. As you load the XML file in Internet Explorer, you will see the resulting XHTML transformation. The connection is indicated in the heading of the XML document with a command like this:

```
<?xml-stylesheet type="text/xsl" href="sampleembedded.xsl"?>
```

This is what I've done in the `sampleembedded.xml` file available in the XslEmbed folder. The related XSL embeds various XSL snippets that I don't have space to discuss in detail. For example, it grabs the entire list of authors or filters a specific group of them with this code:

```
<h2>All Authors</h2>
<ul>
  <xsl:for-each select="books//author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
<h3>E-Authors</h3>
<ul>
  <xsl:for-each select="books/ebook/author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
```

More complex code is used to extract nodes only when a specific value is present in a subnode or attribute, regardless of the higher-level nodes. The following XSL snippet also has an `if` statement and produces an attribute in the resulting node, as a way to build an `href` hyperlink in the HTML:

```
<h3>Marco's works (books + ebooks)</h3>
<ul>
```

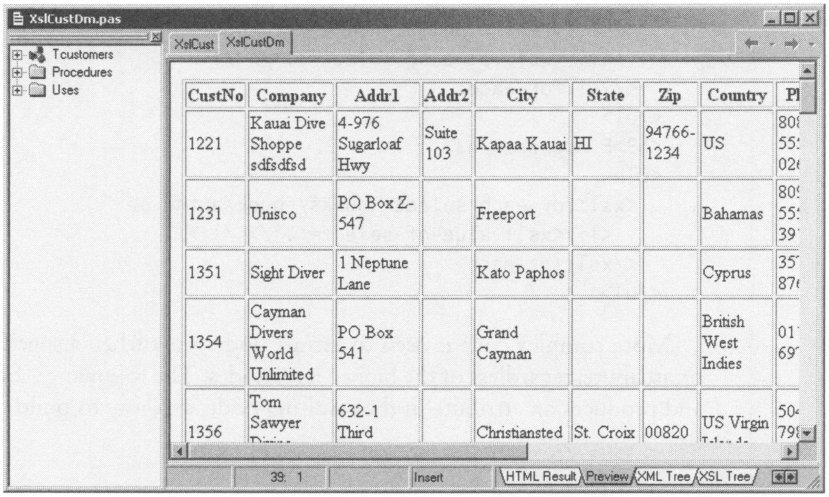
```
    <xsl:for-each select="books/*[author = 'Cantu']">
      <li> <xsl:value-of select="title"/>
          <xsl:if test="url">
            (<a><xsl:attribute name="href"><xsl:value-of select="url"/>
               </xsl:attribute>Jump to document</a>)
          </xsl:if>
        </li>
    </xsl:for-each>
  </ul>
```

XSLT with WebSnap

Within the code of a program, you can execute the `TransformNode` method of a DOM node, passing to it another DOM hosting the XSL document. Instead of using this low-level approach, however, you can let WebSnap help you to create an XSL-based example. You can create a new WebSnap application (I've built a CGI program called XslCust in this case) and choose an XSLPageProducer component for its main page, to let Delphi help you begin the application code. Delphi also includes a skeleton XSL file for manipulating a ClientDataSet data packet and adds many new views to the editor. The XSL text replaces the HTML file; the XML Tree page shows the data, if any; the XSL Tree page shows the XSL within the Internet Explorer ActiveX; the HTML Result page shows the code produced by the transformation; and the Preview page shows what a user will see in a browser.

TIP *In Delphi 7, the editor provides full-blown code completion for XSLT, which makes editing this code in the editor as powerful as it is in some sophisticated and specific XML editors.*

To make this example work, you must provide data to the XSLPageProducer component via its `XMLData` property. This property can be hooked up to an XMLDocument or directly to an XMLBroker component, as I've done in this case. The broker takes its data from a provider connected to a local table attached to the classic `Customer.cds` component table of the classic DBDEMOS. The effect is that, with the following Delphi-generated XSL, you get (even at design time) the output of shown in Figure 22.12:



CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Pl
1221	Kauai Dive Shoppe sdfsdfsd	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	80 55 02
1231	Unisco	PO Box Z-547		Freeport			Bahamas	80 55 39
1351	Sight Diver	1 Neptune Lane		Kato Paphos			Cyprus	35 87
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman			British West Indies	01 69
1356	Tom Sawyer	632-1 Third		Christiansted	St. Croix	00820	US Virgin	50 79

FIGURE 22.12
The result of an XSLT transformation generated (even at design time) by the XSLPageProducer component in the XslCust example

```
<?xml version="1.0"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="DATAPACKET">
    <table border="1">
      <xsl:apply-templates select="METADATA/FIELDS"/>
      <xsl:apply-templates select="ROWDATA/ROW"/>
    </table>
  </xsl:template>

  <xsl:template match="FIELDS">
    <tr>
      <xsl:apply-templates/>
    </tr>
  </xsl:template>

  <xsl:template match="FIELD">
    <th>
      <xsl:value-of select="@attrname"/>
    </th>
  </xsl:template>

  <xsl:template match="ROWDATA/ROW">
  <xsl:variable name="fieldDefs" select="//METADATA/FIELDS"/>
  <xsl:variable name="currentRow" select="current()"/>
  <tr>
    <xsl:for-each select="$fieldDefs/FIELD">
      <td>
         <xsl:value-of select="$currentRow/@*[name()=current()/@attrname]"/><br/>
      </td>
    </xsl:for-each>
  </tr>
  </xsl:template>

</xsl:stylesheet>
```

NOTE The standard XSL template has been extended since Delphi 6, because the original versions didn't account for null fields omitted from the XML data packet. I presented several extensions to the original XSL code at the 2002 Borland Conference, and some of my suggestions have been incorporated in the template.

This code generates an HTML table consisting of the expansion of field metadata and row data. The fields are used to generate the table heading, with a `<th>` cell for each entry in a single row. The row data is used to fill in the other rows of the table. Taking the value of each attribute (`select="@*"`) wouldn't be enough, because an attribute might be missing. For this reason, the list of fields and the current row are saved in two variables; then, for each field, the XSL code extracts the value of a row item having an attribute name (`@*[name()=...`) corresponding to the name of the current field stored in its *attrname* attribute (`@attrname`). This code is far from simple, but it is a compact and portable way to examine different portions of an XML document at the same time.

Direct XSL Transformations with the DOM

Using the XSLPageProducer can be handy, but generating multiple pages based on the same data just to handle different possible XSL styles with WebSnap isn't the best approach. I've built a plain CGI application called CdsXslt that can transform a ClientDataSet data packet into different types of HTML, depending on the name of the XSL file passed as a parameter. The advantage is that I can modify the existing XSL files and add new XSL files to the system without having to recompile the program.

To obtain the XSL transformation, the program loads both the XML and the XSL files into two XMLDocument components called `xmlDom` and `XslDom`. Then it invokes the `transformNode` method of the XML document, passing the XSL document as a parameter and filling in a third XMLDocument component called `HtmlDom`:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  xslfile, xslfolder: string;
  attr: IDOMAttr;
begin
  // open the client dataset and load its XML in a DOM
  ClientDataSet1.Open;
  XmlDom.Xml.Text := ClientDataSet1.XMLData;
  XmlDom.Active := True;
  // load the requested xsl file
  xslfile := Request.QueryFields.Values ['style'];
  if xslfile = '' then
    xslfile := 'customer.xsl';
  xslfolder := ExtractFilePath (ParamStr (0)) + 'xsl\';
  if FileExists (xslfolder + xslfile) then
    xslDom.LoadFromFile (xslfolder + xslfile)
  else
    raise Exception.Create('Missing file: ' + xslfolder + xslfile);
  XSLDom.Active := True;
  if xslfile = 'single.xsl' then
  begin
    attr := xslDom.DOMDocument.createAttribute('select');
    attr.value := '//ROW[@CustNo="' + Request.QueryFields.Values ['id'] + '"]';
    xslDom.DOMDocument.getElementsByTagName ('xsl:apply-templates').
      item[0].attributes.setNamedItem(attr);
  end;
  // do the transformation
  HTMLDom.Active := True;
  xmlDom.DocumentElement.transformNode (xslDom.DocumentElement, HTMLDom);
  Response.Content := HTMLDom.XML.Text;
end;
```

The code uses the DOM to modify the XSL document for displaying a single record, adding the XPath statement for selecting the record indicated by the `id` query field. This `id` is added to the hyperlink by the XSL with the list of records, but I'll skip listing more XSL files. They are available for study in the XSL subfolder of this example's folder.

WARNING *To run this program, deploy the XSL files in a folder called XSL under the one where the script is located. You can find the demo files in the XSL subfolder of the scripts folder of this chapter. To deploy these files in a different location, change the code above that extracts the XSL folder name from the program name available in the first common line parameter (as the global* `Application` *object defined in the Forms unit is not accessible in a CGI application).*

Processing Large XML Documents

As you have seen, there are often many different techniques to accomplish the same task with XML. In many cases you can choose any solution with the goal of writing less and more maintainable code; but when you need to process a large number of XML documents or very large XML documents, you must consider efficiency.

Discussing theory by itself is not terribly useful, so I've built an example you can use (and modify) to test different solutions. The example is called LargeXml, and it covers a specific area: moving data from a database to an XML file and back. The example can open a dataset (using dbExpress) and then replicate the data many times in a ClientDataSet in memory. The structure of the in-memory ClientDataSet is cloned from that of the data access component:

```
SimpleDataSet1.Open;
ClientDataSet1.FieldDefs := SimpleDataSet1.FieldDefs;
ClientDataSet1.CreateDataSet;
```

After using a radio group to determine the amount of data you want to process (some options require minutes on a slow computer), the data is cloned with this code:

```
while ClientDataSet1.RecordCount < nCount do
begin
  SimpleDataSet1.RecNo := Random (SimpleDataSet1.RecordCount) + 1;
  ClientDataSet1.Insert;
  ClientDataSet1.Fields [0].AsInteger := Random (10000);
```

```
    for I := 1 to SimpleDataSet1.FieldCount - 1 do
      ClientDataSet1.Fields [i].AsString := SimpleDataSet1.Fields [i].AsString;
    ClientDataSet1.Post;
  end;
```

From a ClientDataSet to an XML Document

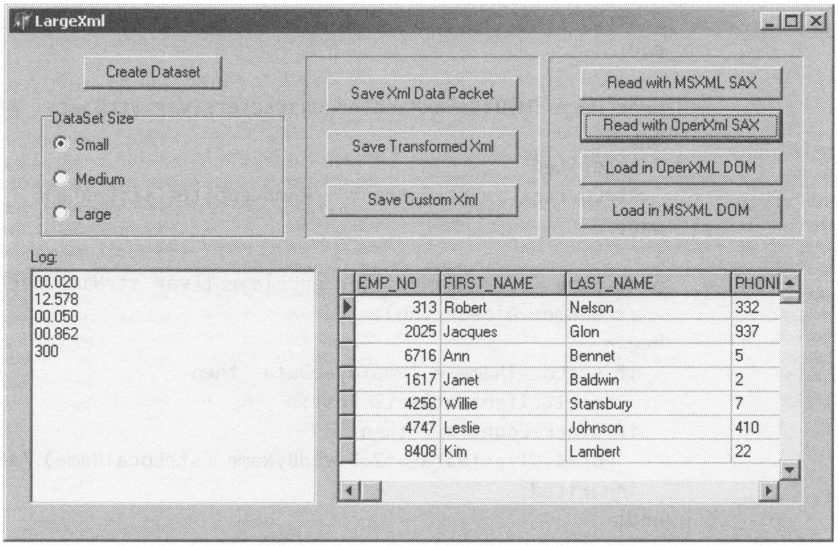
Now that the program has a (large) dataset in memory, it provides three different ways to save the dataset to a file. The first is to save the `XMLData` of the ClientDataSet directly to a file, obtaining an attribute-based document. This is probably not the format you want, so the second solution is to apply an XmlMapper transformation with an XMLTransformClient component. The third solution involves processing the dataset directly and writing out each record to a file:

```
procedure TForm1.btnSaveCustomClick(Sender: TObject);
var
  str: TFileStream;
  s: string;
  i: Integer;
begin
  str := TFileStream.Create ('data3.xml', fmCreate);
  try
    ClientDataSet1.First;
    s := '<?xml version="1.0" standalone="yes" ?><employee>';
    str.Write(s[1], Length (s));

    while not ClientDataSet1.EOF do
    begin
      s := '';
      for i := 0 to ClientDataSet1.FieldCount - 1 do
        s := s + MakeXmlstr (ClientDataSet1.Fields[i].FieldName,
          ClientDataSet1.Fields[i].AsString);
      s := MakeXmlStr ('employeeData', s);
      str.Write(s[1], length (s));
      ClientDataSet1.Next
    end;
    s := '</employee>';
    str.Write(s[1], length (s));
  finally
    str.Free;
  end;
end;
```

This code uses a simple (but effective) support function to create XML nodes:

```
function MakeXmlstr (node, value: string): string;
begin
  Result := '<' + node + '>' + value + '</' + node + '>';
end;
```

If you run the program, you can see the time taken by each operation, as shown in Figure 22.13. Saving the ClientDataSet data is the fastest approach, but you probably don't get the result you want. Custom streaming is only slightly slower; but you should consider that this code doesn't require you to first move the data to a ClientDataSet, because you can apply it directly even to a unidirectional dbExpress dataset. You should forget using the code based on the XmlMapper for a large dataset, because it is hundreds of times slower, even for a small dataset (I haven't been able to try a large dataset, because the process takes too long). For example, the 50 milliseconds required by custom streaming for a small dataset become more than 10 seconds when I use the mapping, and the result is very similar.



DataSet Size
● Small
○ Medium
○ Large

Log:

```
00.020
12.578
00.050
00.862
300
```

EMP_NO	FIRST_NAME	LAST_NAME	PHONI
313	Robert	Nelson	332
2025	Jacques	Glon	937
6716	Ann	Bennet	5
1617	Janet	Baldwin	2
4256	Willie	Stansbury	7
4747	Leslie	Johnson	410
8408	Kim	Lambert	22

FIGURE 22.13
The LargeXml example in action

From an XML Document to a ClientDataSet

Once you have a large XML document, obtained by a program (as in this case) or from an external source, you need to process it. As you have seen, XmlMapper support is far too slow, so you are left with three alternatives: an XSL transformation, a SAX, or a DOM. XSL transformations will probably be fast enough, but in this example I've opened the document with a SAX; it's the fastest approach and doesn't require much code. The program can also load a document in a DOM, but I haven't written the code to navigate the DOM and save the data back to a ClientDataSet.

In both cases, I've tested the OpenXml engine versus the MSXML DOM. This allows you to see the two SAX solutions compared, because (unluckily) the code is slightly different. I can summarize the results here: Using the MSXML SAX is slightly faster than using the OpenXml SAX (the difference is about 20 percent), whereas loading in the DOM marks a large advantage in favor of MSXML.

The MSXML SAX code uses the same architecture discussed in the SaxDemo1 example, so here I've listed only the code of the handlers you use. As you can see, at the beginning of an employeeData element you insert a new record, which is posted when the same node is closed. Lower-level nodes are added as fields of the current record. Here is the code:

```
procedure TMyDataSaxHandler.startElement(var strNamespaceURI, strLocalName,
  strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
  inherited;
  if strLocalName = 'employeeData' then
    Form1.clientdataset2.Insert;
  strCurrent := '';
end;

procedure TMyDataSaxHandler.characters(var strChars: WideString);
begin
  inherited;
  strCurrent := strCurrent + RemoveWhites(strChars);
end;

procedure TMyDataSaxHandler.endElement(var strNamespaceURI, strLocalName,
  strQName: WideString);
begin
  if strLocalName = 'employeeData' then
    Form1.clientdataset2.Post;
  if stack.Count > 2 then
    Form1.ClientDataSet2.FieldByName (strLocalName).AsString := strCurrent;
  inherited;
end;
```

The code for the event handlers in the OpenXml version is similar. All that changes are the interface of the methods and the names of the parameters:

```
type
  TDataSaxHandler = class (TXmlStandardHandler)
  protected
    stack: TStringList;
    strCurrent: string;
  public
    constructor Create(aowner: TComponent); override;
    function endElement(const sender: TXmlCustomProcessorAgent;
      const locator: TdomStandardLocator;
      namespaceURI, tagName: wideString): TXmlParserError; override;
    function PCDATA(const sender: TXmlCustomProcessorAgent;
      const locator: TdomStandardLocator; data: wideString):
      TXmlParserError; override;
    function startElement(const sender: TXmlCustomProcessorAgent;
      const locator: TdomStandardLocator; namespaceURI, tagName: wideString;
      attributes: TdomNameValueList): TXmlParserError; override;
    destructor Destroy; override;
  end;
```

It is also more difficult to invoke the SAX engine, as shown in the following code (from which I've removed the code for the creation of the dataset, the timing, and the logging):

```
procedure TForm1.btnReadSaxOpenClick(Sender: TObject);
var
  agent: TXmlStandardProcessorAgent;
  reader: TXmlStandardDocReader;
  filename: string;
begin
  Log := memoLog.Lines;
  filename := ExtractFilePath (Application.Exename) + 'data3.xml';
  agent := TXmlStandardProcessorAgent.Create(nil);
  reader:= TXmlStandardDocReader.Create (nil);
  try
    reader.NextHandler := TDataSaxHandler.Create (nil); // our custom class
    agent.reader := reader;
    agent.processFile(filename, filename);
  finally
    agent.free;
    reader.free;
  end;
end;
```

What's Next?

In this chapter I've covered XML and related technologies, including DOM, SAX, XSLT, XML schemas, XPath, and a few more. You've seen how Delphi simplifies DOM programming with XML access using interfaces and XML transformations. I've also discussed the use of XSL for web programming, introducing the XSLT support of WebSnap and the Internet Express architecture.

Chapter 23 will continue the discussion of XML with one of the most interesting and promising technologies of the last few years: web services. I'll cover SOAP and WSDL, and also introduce UDDI and other related technologies. If you are interested in the topics discussed here from the Delphi perspective, you should refer to books specifically devoted to XML, XML schemas, and XSLT.

Chapter 23

Web Services and SOAP

OF ALL THE RECENT features of Delphi, one stands out clearly: the support for web services built into the product. The fact that I'm discussing it near the end of the book has nothing to do with its importance, but only with the logical flow of the text, and with the fact that this is not the starting point from which to learn Delphi programming.

The topic of web services is broad and involves many technologies and business-related standards. As usual, I'll focus on the underlying Delphi implementation and the technical side of web services, rather than discuss the larger picture and business implications.

This chapter is also relevant because Delphi 7 adds a lot of power to the implementation of web services provided by Delphi 6, including support for attachments, custom headers, and much more. You'll see how to create a web service client and a web service server, and also how to move database data over SOAP using the DataSnap technology.

This chapter covers the following topics:

- Web services
- SOAP and WSDL
- DataSnap over SOAP
- Handling attachments
- UDDI

Web Services

The rapidly emerging web services technology has the potential to change the way the Internet works for businesses. Browsing web pages to enter orders is fine for individuals (business-to-consumer [B2C] applications) but not for companies (business-to-business [B2B] applications). If you want to buy a few books, going to a book vendor website and punching in your requests is probably fine. But if you run a bookstore and want to place hundreds of orders a day, this is far from a good approach, particularly if you have a program that helps you track your sales and determine reorders. Grabbing the output of this program and reentering it into another application is ridiculous.

Web services are meant to solve this issue: The program used to track sales can automatically create a request and send it to a web service, which can immediately return information about the order. The next step might be to ask for a tracking number for the shipment. At this point, your program can use another web service to track the shipment until it is at its destination, so you can tell your customers how long they have to wait. As the shipment arrives, your program can send a reminder via SMS or pager to the people with pending orders, issue a payment with a bank web service, and ... I could continue but I think I've given you the idea. Web services are meant for computer interoperability, much as the Web and e-mail let people interact.

SOAP and WSDL

Web services are made possible by the Simple Object Access Protocol (SOAP). SOAP is built over standard HTTP, so that a web server can handle the SOAP requests and the related data packets can pass though firewalls. SOAP defines an XML-based notation for requesting the execution of a method by an object on the server and passing parameters to it; another notation defines the format of a response.

NOTE *SOAP was originally developed by DevelopMentor (the training company run by COM expert Don Box) and Microsoft, to overcome weaknesses involved with using DCOM in web servers. Submitted to the W3C for standardization, it is being embraced by many companies, with a particular push from IBM. It is too early to know whether there will be standardization to let software programs from Microsoft, IBM, Sun, Oracle, and many others truly interoperate, or whether some of these vendors will try to push a private version of the standard. In any case, SOAP is a cornerstone of Microsoft's .NET architecture and also of the current platforms by Sun and Oracle.*

SOAP will replace COM invocation, at least between different computers. Similarly, the definition of a SOAP service in the Web Services Description Language (WSDL) format will replace the IDL and type libraries used by COM and COM+. WSDL documents are another type of XML document that provides the metadata definition of a SOAP request. As you get a file in this format (generally published to define a service), you'll be able to create a program to call it.

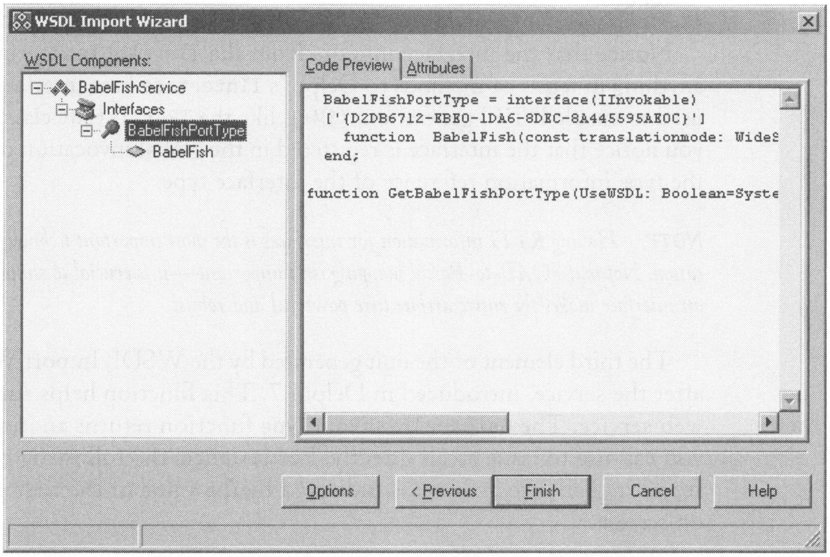
Specifically, Delphi provides a bi-directional mapping between WSDL and interfaces. This means you can grab a WSDL file and generate an interface for it. You can then create a client program, embedding SOAP requests via these interfaces, and use a special Delphi component that lets you convert your local interface requests into SOAP calls (I doubt you want to manually generate the XML required for a SOAP request).

The other way around, you can define an interface (or use an existing one) and let a Delphi component generate a WSDL description for it. Another component provides you with a SOAP-to-Pascal mapping, so that by embedding this component and an object implementing the interface within a server-side program, you can have your web service up and running in a matter of minutes.

BabelFish Translations

As a first example of the use of web service, I've built a client for the BabelFish translation service offered by AltaVista. You can find this and many other services for experimentation on the XMethods website (`www.xmethods.com`).

After downloading the WSDL description of this service from XMethods (also available among the source code files for this chapter), I invoked Delphi's Web Services Importer in the Web Services page of the New Items dialog box and selected the file. The wizard lets you preview the structure of the service (see Figure 23.1) and generate the proper Delphi-language interfaces in a unit like the following (with many of the comments removed):


WSDL Import Wizard
WSDL Components:
BabelFishService
Interfaces
BabelFishPortType
BabelFish
Code Preview
Attributes
Options
< Previous
Finish
Cancel
Help

```
BabelFishPortType = interface(IInvokable)
['{D2DB6712-EBE0-1DA6-8DEC-8A445595AE0C}']
  function  BabelFish(const translationmode: WideS
end;

function GetBabelFishPortType(UseWSDL: Boolean=Syste
```

FIGURE 23.1
The WSDL Import Wizard in action

```
unit BabelFishService;

interface

uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;

type
  BabelFishPortType = interface(IInvokable)
  ['{D2DB6712-EBE0-1DA6-8DEC-8A445595AE0C}']
    function  BabelFish(const translationmode: WideString;
      const sourcedata: WideString): WideString; stdcall;
  end;

function GetBabelFishPortType(UseWSDL: Boolean=System.False;
  Addr: string=''; HTTPRIO: THTTPRIO = nil): BabelFishPortType;

implementation

// omitted
```

```
initialization
  InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType),
    'urn:xmethodsBabelFish', '');
  InvRegistry.RegisterDefaultSOAPAction(TypeInfo(BabelFishPortType),
    'urn:xmethodsBabelFish#BabelFish');

end.
```

Notice that the interface inherits from the `IInvokable` interface. This interface doesn't add anything in terms of methods to Delphi's `IInterface` base interface, but it is compiled with the flag used to enable RTTI generation, `{$M+}`, like the `TPersistent` class. In the `initialization` section, you notice that the interface is registered in the global invocation registry (or `InvRegistry`), passing the type information reference of the interface type.

NOTE *Having RTTI information for interfaces is the most important technological advance underlying SOAP invocation. Not that SOAP-to-Pascal mapping isn't important—it is crucial to simplify the process—but having RTTI for an interface makes the entire architecture powerful and robust.*

The third element of the unit generated by the WSDL Import Wizard is a global function named after the service, introduced in Delphi 7. This function helps simplify the code used to call the web service. The `GetBabelFishPortType` function returns an interface of the proper type, which you can use to issue a call directly. For instance, the following code translates a short sentence from English into Italian (as indicated by the value of the first parameter, *en_it*) and shows it on screen:

```
ShowMessage (GetBabelFishPortType.BabelFish('en_it', 'Hello, world!'));
```

If you look at the code for the `GetBabelFishPortType` function, you'll see that it creates an internal invocation component of the class `THTTPRIO` to process the call. You can also place this component manually on the client form (as I've done in the example program) to gain better control over its various settings (and handle its events).

This component can be configured in two basic ways: You can refer to the WSDL file or URL, import it, and extract from it the URL of the SOAP call; or, you can provide a direct URL to call. The example has two components that provide the alternative approaches (with exactly the same effect):

```
object HTTPRIO1: THTTPRIO
  WSDLLocation = 'BabelFishService.xml'
  Service = 'BabelFish'
  Port = 'BabelFishPort'
end
object HTTPRIO2: THTTPRIO
  URL = 'http://services.xmethods.net:80/perl/soaplite.cgi'
end
```

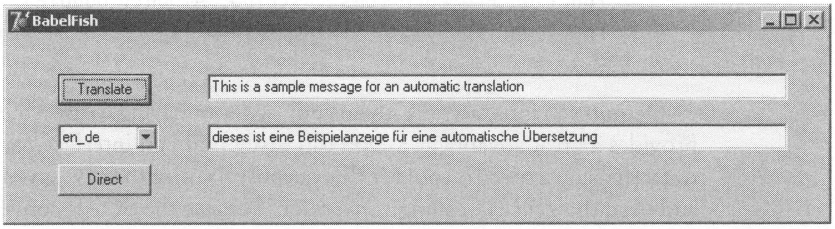
At this point, there is little left to do. You have information about the service that can be used for its invocation, and you know the types of the parameters required by the only available method as they are listed in the interface. The two elements are merged by extracting the interface you want to call directly from the `HTTPRIO` component, with an expression like `HTTPRIO1 as BabelFishPortType`. It might seem astonishing at first, but it is also outrageously simple.

This is the web service call done by the example:

```
EditTarget.Text := (HTTPRIO1 as BabelFishPortType).
  BabelFish(ComboBoxType.Text, EditSource.Text);
```

The program output, shown in Figure 23.2, allows you to learn foreign languages (although the teacher has its shortcomings!). I haven't replicated the same example with stock options, currencies, weather forecasts, and the many other services available, because they would look much the same.

WARNING *Although the web service interface provides you with the types of the parameters, in many cases you need to refer to some actual documentation of the service to know what the values of the parameters really mean and how they are interpreted by the service. The BabelFish web service is an example of this issue, as I had to look at some textual documentation to find out the list of translation types, available in the demo in a combo box.*



This is a sample message for an automatic translation

en_de

dieses ist eine Beispielanzeige für eine automatische Übersetzung

FIGURE 23.2 An example of a translation from English to German obtained by AltaVista's BabelFish via a web service

Building Web Services

If calling a web service in Delphi is straightforward, the same can be said of developing a service. If you go into the Web Services page of the New Items dialog box, you can see the SOAP Server Application option. Select it, and Delphi presents you with a list that's quite similar to what you see if you select a WebBroker application. A web service is typically hosted by a web server using one of the available web server extension technologies (CGI, ISAPI, Apache modules, and so on) or the Web App Debugger for your initial tests.

After completing this step, Delphi adds three components to the resulting web module, which is just a plain web module with no special additions:

- The HTTPSoapDispatcher component receives the web request, as any other HTTP dispatcher does.
- The HTTPSoapPascalInvoker component does the reverse operation of the HTTPRIO component; it can translate SOAP requests into calls of Pascal interfaces (instead of shifting interface method calls into SOAP requests).

- The WSDLHTMLPublish component can be used to extract the WSDL definition of the service from the interfaces it support, and performs the opposite role of the Web Services Importer Wizard. Technically, this is another HTTP dispatcher.

A Currency Conversion Web Service

Once this framework is in place—something you can also do by adding the three components listed in the previous section to an existing web module—you can begin writing a service. As an example, I've taken the euro conversion example from Chapter 3, "The Run-Time Library," and transformed it into a web service called ConvertService. First, I've added to the program a unit defining the interface of the service, as follows:

```
type
  IConvert = interface(IInvokable)
  ['{FF1EAA45-0B94-4630-9A18-E768A91A78E2}']
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

Defining an interface directly in code, without having to use a tool such as the Type Library Editor, provides a great advantage, as you can easily build an interface for an existing class and don't have to learn using a specific tool for this purpose. Notice that I've given a GUID to the interface, as usual, and used the `stdcall` calling convention, because the SOAP converter does not support the default `register` calling convention.

In the same unit that defines the interface of the service, you should also register it. This operation will be necessary on both the client and server sides of the program, because you will be able to include this interface definition unit in both:

```
uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IConvert));
```

Now that you have an interface you can expose to the public, you have to provide an implementation for it, again by means of the standard Delphi code (and with the help of the predefined `TInvokableClass` class:

```
type
  TConvert = class (TInvokableClass, IConvert)
  protected
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

The implementation of these functions, which call the code of the euro conversion system from Chapter 3, is not discussed here because it has little to do with the development of the service. However, it is important to notice that this implementation unit also has a registration call in its initialization section:

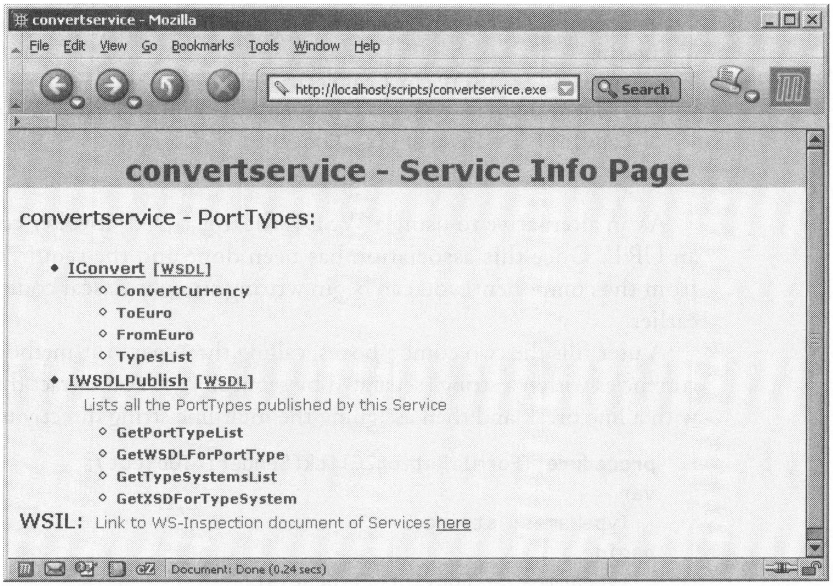
```
InvRegistry.RegisterInvokableClass (TConvert);
```

PUBLISHING THE WSDL

By registering the interface, you make it possible for the program to generate a WSDL description. The web service application (since the Delphi 6.02 update) is capable of displaying a first page describing its interfaces and the detail of each interface, and returning the WSDL file. By connecting to the web service via a browser, you'll see something similar to Figure 23.3.

FIGURE 23.3

The description of the Convert-Service web service provided by Delphi components


convertservice - Mozilla
File
Edit
View
Go
Bookmarks
Tools
Window
Help
http://localhost/scripts/convertservice.exe
Search
Document: Done (0.24 secs)

convertservice - Service Info Page

convertservice - PortTypes:

- IConvert [WSDL]
  - ConvertCurrency
  - ToEuro
  - FromEuro
  - TypesList
- IWSDLPublish [WSDL]
  Lists all the PortTypes published by this Service
  - GetPortTypeList
  - GetWSDLForPortType
  - GetTypeSystemsList
  - GetXSDForTypeSystem

WSIL: Link to WS-Inspection document of Services here

NOTE *Although other web service architectures automatically provide you with a way to execute the web service from the browser, this technique is mostly meaningless, because using web services makes sense in an architecture where different applications interoperate. If all you need to do is show data on a browser, you should build a website!*

This auto-descriptive feature was not available in web services produced in Delphi 6 (which provided only the lower-level WSDL listing), but it is quite easy to add (or customize). If you look at the Delphi 7 SOAP web module you'll notice a default action with an `OnAction` event handler invoking the following default behavior:

```
WSDLHTMLPublish1.ServiceInfo(Sender, Request, Response, Handled);
```

This is all you have to do to retrofit this feature into an existing Delphi web service that lacks it. To provide similar functionality manually, you must call into the invocation registry (the `InvRegistry` global object), with calls like `GetInterfaceExternalName` and `GetMethExternalName`.

What's important is the web service application's ability to document itself to any other programmer or programming tool, by exposing the WSDL.

CREATING A CUSTOM CLIENT

Let's move to the client application that calls the service. I don't need to start from the WSDL file, because I already have the Delphi interface. This time the form doesn't even have the HTTPRIO component, which is created in code:

```
private
  Invoker: THTTPRio;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Invoker := THTTPRio.Create(nil);
  Invoker.URL := 'http://localhost/scripts/ConvertService.exe/soap/iconvert';
  ConvIntf := Invoker as IConvert;
end;
```

As an alternative to using a WSDL file, the SOAP invoker component can be associated with an URL. Once this association has been done and the required interface has been extracted from the component, you can begin writing straight Pascal code to invoke the service, as you saw earlier.

A user fills the two combo boxes, calling the `TypesList` method, which returns a list of available currencies within a string (separated by semicolons). You extract this list by replacing each semicolon with a line break and then assigning the multiline string directly to the combo items:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  TypeNames: string;
begin
  TypeNames := ConvIntf.TypesList;
  ComboBoxFrom.Items.Text := StringReplace (TypeNames, ';', sLineBreak,
    [rfReplaceAll]);
  ComboBoxTo.Items := ComboBoxFrom.Items;
end;
```

After selecting two currencies, you can perform the conversion with this code (Figure 23.4 shows the result):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  LabelResult.Caption := Format ('%n', [(ConvIntf.ConvertCurrency(
    ComboBoxFrom.Text, ComboBoxTo.Text, StrToFloat(EditAmount.Text)))]);
end;
```

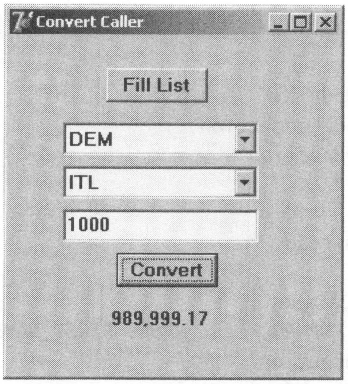


FIGURE 23.4
The ConvertCaller client of the ConvertService web service shows how few German marks you used to get for so many Italian liras, before the euro changed everything.

Asking for Database Data

For this example, I built a web service (based on the Web App Debugger) capable of exposing data about employees of a company. This data is mapped to the EMPLOYEE table of sample InterBase database we've used so often throughout the book. The Delphi interface of the web service is defined in the SoapEmployeeIntf unit as follows:

```
type
  ISoapEmployee = interface (IInvokable)
    ['{77D0D940-23EC-49A5-9630-ADE0751E3DB3}']
    function GetEmployeeNames: string; stdcall;
    function GetEmployeeData (EmpID: string): string; stdcall;
  end;
```

The first method returns a list of the names of all the employees in the company, and the second returns the details of a given employee. The implementation of this interface is provided in the SoapEmployeeImpl unit with the following class:

```
type
  TSoapEmployee = class(TInvokableClass, ISoapEmployee)
  public
    function GetEmployeeNames: string; stdcall;
    function GetEmployeeData (EmpID: string): string; stdcall;
  end;
```

The implementation of the web service lies in the two previous methods and some helper functions to manage the XML data being returned. But before we get to the XML portion of the example, let me briefly discuss the database access section.

ACCESSING THE DATA

All the connectivity and SQL code in this example are hosted in a separate data module. Of course, I could have created some connection and dataset components dynamically in the methods, but doing so is contrary to the approach of a visual development tool like Delphi. The data module has the following structure:

```
object DataModule3: TDataModule3
  object SQLConnection: TSQLConnection
    ConnectionName = 'IBConnection'
    DriverName = 'Interbase'
    LoginPrompt = False
    Params.Strings = // omitted
  end
   object dsEmplList: TSQLDataSet
    CommandText = 'select EMP_NO, LAST_NAME, FIRST_NAME from EMPLOYEE'
    SQLConnection = SQLConnection
    object dsEmplListEMP_NO: TStringField
    object dsEmplListLAST_NAME: TStringField
    object dsEmplListFIRST_NAME: TStringField
  end
  object dsEmpData: TSQLDataSet
    CommandText = 'select * from EMPLOYEE where Emp_No = :id'
    Params = <
      item
        DataType = ftFixedChar
        Name = 'id'
        ParamType = ptInput
      end>
    SQLConnection = SQLConnection
   end
end
```

As you can see, the data module has two SQL queries hosted by SQLDataSet components. The first is used to retrieve the name and ID of each employee, and the second returns the entire set of data for a given employee.

PASSING XML DOCUMENTS

The problem is how to return this data to a remote client program. In this example, I've used the approach I like best: I've returned XML documents, instead of working with complex SOAP data structures. (I don't get how XML can be seen as a messaging mechanism for SOAP invocation—along with the transport mechanism provided by HTTP—but then, it is not used for the data being transferred. Still, very few web services return XML documents, so I'm beginning to wonder if it's me or many other programmers who can't see the full picture.)

In this example, the `GetEmployeeNames` method creates an XML document containing a list of employees, with their first and last names as values and the related database ID as an attribute, using two helper functions `MakeXmlStr` (already described in the last chapter) and `MakeXmlAttribute` (listed here):

```
function TSoapEmployee.GetEmployeeNames: string;
var
  dm: TDataModule3;
```

```
begin
  dm := TDataModule3.Create (nil);
  try
    dm.dsEmplList.Open;
    Result := '<employeeList>' + sLineBreak;
    while not dm.dsEmplList.EOF do
    begin
      Result := Result + '  ' + MakeXmlStr ('employee',
        dm.dsEmplListLASTNAME.AsString + ' ' +
        dm.dsEmplListFIRSTNME.AsString,
        MakeXmlAttribute ('id', dm.dsEmplListEMPNO.AsString)) + sLineBreak;
      dm.dsEmplList.Next;
    end;
    Result := Result + '</employeeList>';
  finally
    dm.Free;
  end;
end;

function MakeXmlAttribute (attrName, attrValue: string): string;
begin
  Result := attrName + '="' + attrValue + '"';
end;
```

Instead of the manual XML generation, I could have used the XML Mapper or some other technology; but as you should know from Chapter 22 ("Using XML Technologies"), I rather prefer creating XML directly in strings. I'll use the XML Mapper to process the data received on the client side.

NOTE *You may wonder why the program creates a new instance of the data module each time. The negative side of this approach is that each time, the program establishes a new connection to the database (a rather slow operation); but the plus side is that you have no risk related to the use of a multithreaded application. If two web service requests are executed concurrently, you can use a shared connection to the database, but you must use different dataset components for the data access. You could move the datasets in the function code and keep only the connection on the data module, or have a global shared data module for the connection (used by multiple threads) and a specific instance of a second data module hosting the datasets for each method call.*

Let's now look at the second method, `GetEmployeeData`. It uses a parametric query and formats the resulting fields in separate XML nodes (using another helper function, `FieldsToXml`):

```
function TSoapEmployee.GetEmployeeData(EmpID: string): string;
var
  dm: TDataModule3;
begin
  dm := TDataModule3.Create (nil);
  try
    dm.dsEmpData.ParamByName('ID').AsString := EmpId;
    dm.dsEmpData.Open;
```

```
      Result := FieldsToXml ('employee', dm.dsEmpData);
    finally
      dm.Free;
    end;
  end;

  function FieldsToXml (rootName: string; data: TDataSet): string;
  var
    i: Integer;
  begin
    Result := '<' + rootName + '>' + sLineBreak;;
    for i := 0 to data.FieldCount - 1 do
      Result := Result + '  ' + MakeXmlStr (
        LowerCase (data.Fields[i].FieldName),
        data.Fields[i].AsString) + sLineBreak;
    Result := Result + '</' + rootName + '>' + sLineBreak;;
  end;
```

THE CLIENT PROGRAM (WITH XML MAPPING)

The final step for this example is to write a test client program. You can do so as usual by importing the WSDL file defining the web service. In this case, you also have to convert the XML data you receive into something more manageable—particularly the list of employees returned by the `GetEmployeeNames` method. As mentioned earlier, I've used Delphi's XML Mapper to convert the list of employees received from the web service into a dataset I can visualize using a DBGrid.

To accomplish this, I first wrote the code to receive the XML with the list of employees and copied it into a memo component and from there to a file. Then, I opened the XML Mapper, loaded the file, and generated from it the structure of the data packet and the transformation file. (You can find the transformation file among the source code files of the SoapEmployee example.) To show the XML data within a DBGrid, the program uses an XMLTransformProvider component, referring to the transformation file:

```
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'EmplListToDataPacket.xtr'
end
```

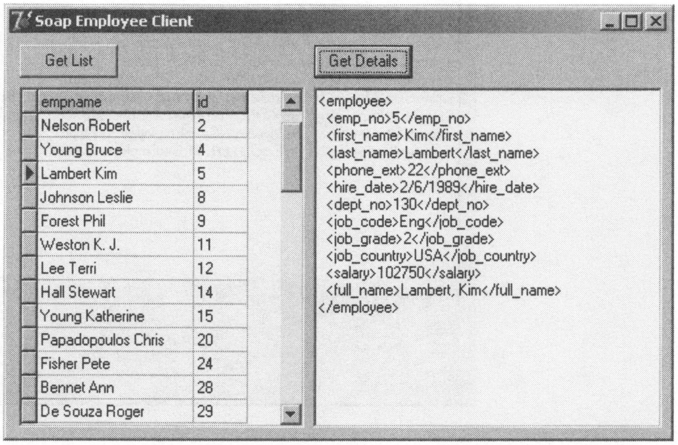
The ClientDataSet component is not hooked to the provider, because it would try to open the XML data file specified by the transformation. In this case, the XML data doesn't reside in a file, but is passed to the component after calling the web service. For this reason the program moves the data to the ClientDataSet directly in code:

```
procedure TForm1.btnGetListClick(Sender: TObject);
var
  strXml: string;
begin
  strXml := GetISoapEmployee.GetEmployeeNames;
  strXML := XMLTransformProvider1.TransformRead.TransformXML(strXml);
```

```
    ClientDataSet1.XmlData := strXml;
    ClientDataSet1.Open;
  end;
```

With this code, the program can display the list of employees in a DbGrid, as you can see in Figure 23.5. When you retrieve the data for the specific employee, the program extracts the ID of the active record from the ClientDataSet and then shows the resulting XML in a memo:

```
procedure TForm1.btnGetDetailsClick(Sender: TObject);
begin
  Memo2.Lines.Text := GetISoapEmployee.GetEmployeeData(
    ClientDataSet1.FieldByName ('id').AsString);
end;
```



empname	id
Nelson Robert	2
Young Bruce	4
Lambert Kim	5
Johnson Leslie	8
Forest Phil	9
Weston K. J.	11
Lee Terri	12
Hall Stewart	14
Young Katherine	15
Papadopoulos Chris	20
Fisher Pete	24
Bennet Ann	28
De Souza Roger	29

```
<employee>
  <emp_no>5</emp_no>
  <first_name>Kim</first_name>
  <last_name>Lambert</last_name>
  <phone_ext>22</phone_ext>
  <hire_date>2/6/1989</hire_date>
  <dept_no>130</dept_no>
  <job_code>Eng</job_code>
  <job_grade>2</job_grade>
  <job_country>USA</job_country>
  <salary>102750</salary>
  <full_name>Lambert, Kim</full_name>
</employee>
```

FIGURE 23.5
The client program of the SoapEmployee web service example

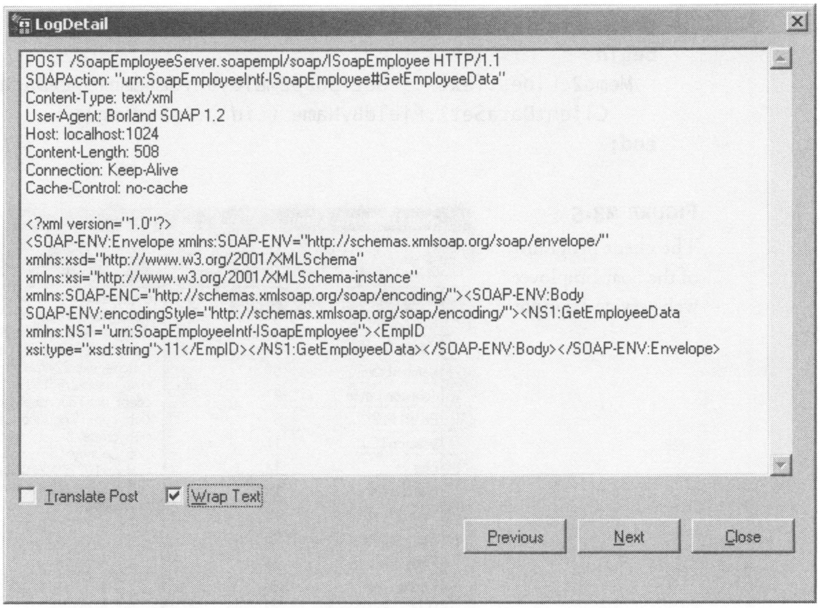
Debugging the SOAP Headers

One final note for this example relates to the use of the Web App Debugger for testing SOAP applications. Of course, you can run the server program from the Delphi IDE and debug it easily, but you can also monitor the SOAP headers passed on the HTTP connection. Although looking at SOAP from this low-level perspective can be far from simple, it is the ultimate way to check if something is wrong with either a server or a client SOAP application. As an example, in Figure 23.6 you can see the HTTP log of a SOAP request from the last example.

The Web App Debugger might not always be available, so another common technique is to handle the events of the HTTPRIO component, as the BabelFishDebug example does. The program's form has two memo components in which you can see the SOAP request and the SOAP response:

```
procedure TForm1.HTTPRIO1BeforeExecute(const MethodName: String;
  var SOAPRequest: WideString);
begin
  MemoRequest.Text := SoapRequest;
end;
```

```
procedure TForm1.HTTPRIO1AfterExecute(const MethodName: String;
  SOAPResponse: TStream);
begin
  SOAPResponse.Position := 0;
  MemoResponse.Lines.LoadFromStream(SOAPResponse);
end;
```



```
POST /SoapEmployeeServer.soapempl/soap/ISoapEmployee HTTP/1.1
SOAPAction: "urn:SoapEmployeeIntf-ISoapEmployee#GetEmployeeData"
Content-Type: text/xml
User-Agent: Borland SOAP 1.2
Host: localhost:1024
Content-Length: 508
Connection: Keep-Alive
Cache-Control: no-cache

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"><SOAP-ENV:Body
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><NS1:GetEmployeeData
xmlns:NS1="urn:SoapEmployeeIntf-ISoapEmployee"><EmpID
xsi:type="xsd:string">11</EmpID></NS1:GetEmployeeData></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

FIGURE 23.6
The HTTP log of the Web App Debugger includes the low-level SOAP request.

Exposing an Existing Class as a Web Service

Although you might want to begin developing a web service from scratch, in some cases you may have existing code to make available. This process is not too complex, given Delphi's open architecture in this area. To try it, follow these steps:

1. Create a web service application or add the related components to an existing WebBroker project.
2. Define an interface inheriting from `IInvokable` and add to it the methods you want to make available in the web service (using the `stdcall` calling convention). The methods will be similar to those of the class you want to make available.
3. Define a new class that inherits from the class you want to expose and implements your interface. The methods will be implemented by calling the corresponding methods of the base class.
4. Write a factory method to create an object of your implementation class any time a SOAP request needs it.

This last step is the most complex. You could define a factory and register it as follows:

```
procedure MyObjFactory (out Obj: TObject);
begin
  Obj := TMyImplClass.Create;
end;

initialization
  InvRegistry.RegisterInvokableClass(TMyImplClass, MyObjFactory);
```

However, this code creates a new object for every call. Using a single global object would be equally bad: Many different users might try to use it, and if the object has state or its methods are not concurrent, you might be in for big problems. You're left with the need to implement some form of session management, which is a variation on the problem we had with the earlier web service connecting to the database.

DataSnap over SOAP

Now that you have a reasonably good idea how to build a SOAP server and a SOAP client, let's look at how to use this technology in building a multitier DataSnap application. You'll use a Soap Server Data Module to create the new web service and the SoapConnection component to connect a client application to it.

Building the DataSnap SOAP Server

Let's look at the server side first. Go to the Web Services page of the New Items dialog box and use the Soap Server Application icon to create a new web service, and then use the Soap Server Data Module icon to add a DataSnap server-side data module to the SOAP server. I did this in the SoapDataServer7 example (which uses the Web App Debugger architecture for testing purposes). From this point on, all you do is write a normal DataSnap server (or a middle-tier DataSnap application), as discussed in Chapter 16 ("Multitier DataSnap Applications"). In this case, I added InterBase access to the program by means of dbExpress, resulting in the following structure:

```
object SoapTestDm: TSoapTestDm
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
  end
end
```

The data module built for a SOAP-based DataSnap server defines a custom interface (so you can add methods to it) inheriting from `IAppServerSOAP`, which is defined as a published interface (even though it doesn't inherit from `IInvokable`).

TIP *Delphi 6 used DataSnap's standard* `IAppServer` *interface for exposing data via SOAP. Delphi 7 has replaced the default with the inherited* `IAppServerSOAP` *interface, which is functionally identical but allows the system to discriminate the type of call depending on the interface name. You'll see shortly how to call an older application from a client built in Delphi 7, because this process is not automatic.*

The implementation class, `TSoapTestDm`, is the data module, as in other DataSnap servers. Here is the code Delphi generated, with the addition of the custom method:

```
type
  ISampleDataModule = interface(IAppServerSOAP)
    ['{D47A293F-4024-4690-9915-8A68CB273D39}']
    function GetRecordCount: Integer; stdcall;
  end;

  TSampleDataModule = class(TSoapDataModule, ISampleDataModule,
      IAppServerSOAP, IAppServer)
    DataSetProvider1: TDataSetProvider;
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
  public
    function GetRecordCount: Integer; stdcall;
  end;
```

The base `TSoapDataModule` doesn't inherit from `TInvokableClass`. This is not a problem as long as you provide an extra factory procedure to create the object (which is what `TInvokableClass` does for you) and add it to the registration code (as discussed earlier, in the section "Exposing an Existing Class as a Web Service"):

```
procedure TSampleDataModuleCreateInstance(out obj: TObject);
begin
  obj := TSampleDataModule.Create(nil);
end;

initialization
  InvRegistry.RegisterInvokableClass(
    TSampleDataModule, TSampleDataModuleCreateInstance);
  InvRegistry.RegisterInterface(TypeInfo(ISampleDataModule));
```

The server application also publishes the `IAppServerSOAP` and `IAppServer` interfaces, thanks to the (little) code in the SOAPMidas unit. As a comparison, you can find a SOAP DataSnap server built with Delphi 6 in the SoapDataServer folder. The example can still be compiled in Delphi 7 and works fine, but you should write new programs using the structure of the newer; an example is in the SoapDataServer7 folder.

TIP *Web service applications in Delphi 7 can include more than one SOAP data module. To identify a specific SOAP data module, use the* `SOAPServerIID` *property of the SoapConnection component or add the data module interface name to the end of the URL.*

The server has a custom method (the Delphi 6 version of the program also had one, but it never worked) that uses a query with the `select count(*) from EMPLOYEE` SQL statement:

```
function TSampleDataModule.GetRecordCount: Integer;
begin
  // read in the record count by running a query
  SQLDataSet2.Open;
  Result := SQLDataSet2.Fields[0].AsInteger;
  SQLDataSet2.Close;
end;
```

Building the DataSnap SOAP Client

To build the client application, called SoapDataClient7, I began with a plain program and added a SoapConnection component to it (from the Web Services page of the palette), hooking it to the URL of the DataSnap web service and referring to the specific interface I'm looking for:

```
object SoapConnection1: TSoapConnection
  URL = 'http://localhost:1024/SoapDataServer7.soapdataserver/' +
    'soap/Isampledatamodule'
  SOAPServerIID = 'IAppServerSOAP - {C99F4735-D6D2-495C-8CA2-E53E5A439E61}'
  UseSOAPAdapter = False
end
```

Notice the last property, `UseSOAPAdapter`, which indicates you are working against a server built with Delphi 7. As a comparison, the SoapDataClient (again, no 7) example, which uses a server created with Delphi 6 and recompiled with Delphi 7, must have this property set to True. This value forces the program to use the plain `IAppServer` interface instead of the new `IAppServerSOAP` interface.

From this point on, you proceed as usual, adding a ClientDataSet component, a DataSource, and a DBGrid to the program; choosing the only available provider for the client dataset; and hooking the rest. Not surprisingly, for this simple example, the client application has little custom code: a single call to open the connection when a button is clicked (to avoid startup errors) and an `ApplyUpdates` call to send changes back to the database.

SOAP versus Other DataSnap Connections

Regardless of the apparent similarity of this program to all the other DataSnap client and server programs built in Chapter 16, there is a very important difference worth underlining: The SoapDataServer and SoapDataClient programs do not use COM to expose or call the `IAppServerSOAP` interface. Quite the opposite—the socket- and HTTP-based connections of DataSnap still rely on local COM objects and a registration of the server in the Windows Registry. The native SOAP-based support, however, allows for a totally custom solution that's independent of COM and that offers many more chances to be ported to other operating systems. (You can recompile this server in Kylix, but not the programs built in Chapter 16.)

The client program can also call the custom method I've added to the server to return the record count. This method could be used in a real-world application to show only a limited number of records but inform the user how many haven't yet been downloaded from the server. The client code to call the method relies on an extra HTTPRIO component:

```
procedure TFormSDC.Button3Click(Sender: TObject);
var
  SoapData: ISampleDataModule;
begin
  SoapData := HttpRio1 as ISampleDataModule;
  ShowMessage (IntToStr (SoapData.GetRecordCount));
end;
```

Handling Attachments

One of the most important features Borland added to Delphi 7 is full support for SOAP attachments. Attachments in SOAP allow you to send data other than XML text, such as binary files or images. In Delphi, attachments are managed through streams. You can read or indicate the type of attachment encoding, but the transformation of a raw stream of bytes into and from a given encoding is up to your code. This process isn't too complex, though, if you consider that Indy includes a number of encoding components.

As an example of how to use attachments, I've written a program that forwards the binary content of a ClientDataSet (also hosts images) or one of the images alone. The server has the following interface:

```
type
  ISoapFish = interface(IInvokable)
  ['{4E4C57BF-4AC9-41C2-BB2A-64BCE470D450}']
    function GetCds: TSoapAttachment; stdcall;
    function GetImage(fishName: string): TSoapAttachment; stdcall;
  end;
```

The implementation of the `GetCds` method uses a ClientDataSet that refers to the classic BIOLIFE table, creates a memory stream, copies the data to it, and then attaches the stream to the `TSoapAttachment` result:

```
function TSoapFish.GetCds: TSoapAttachment; stdcall;
var
  memStr: TMemoryStream;
begin
  Result := TSoapAttachment.Create;
  memStr := TMemoryStream.Create;
  WebModule2.cdsFish.SaveToStream(MemStr); // binary
  Result.SetSourceStream (memStr, soReference);
end;
```

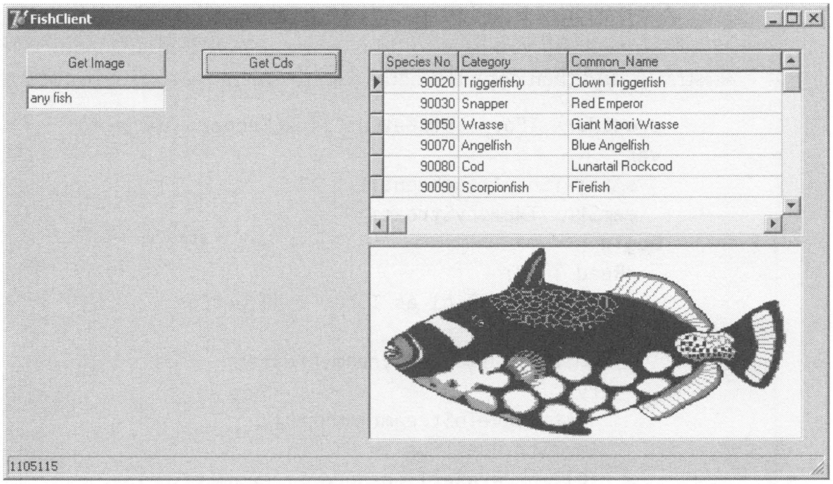
On the client side, I prepared a form with a ClientDataSet component connected to a DBGrid and a DBImage. All you have to do is grab the SOAP attachment, save it to a temporarily in-memory stream, and then copy the data from the memory stream to the local ClientDataSet:

```
procedure TForm1.btnGetCdsClick(Sender: TObject);
var
  sAtt: TSoapAttachment;
  memStr: TMemoryStream;
begin
  nRead := 0;
  sAtt := (HttpRio1 as ISoapFish).GetCds;
  try
    memStr := TMemoryStream.Create;
    try
      sAtt.SaveToStream(memStr);
      memStr.Position := 0;
      ClientDataSet1.LoadFromStream(MemStr);
    finally
      memStr.Free;
    end;
  finally
    DeleteFile (sAtt.CacheFile);
    sAtt.Free;
  end;
end;
```

WARNING *By default, SOAP attachments received by a client are saved to a temporary file, referenced by the* `CacheFile` *property of the* `TSOAPAttachment` *object. If you don't delete this file it will remain in a folder that hosts temporary files.*

This code produces the same visual effect as a client application loading a local file into a ClientDataSet, as you can see in Figure 23.7. In this SOAP client I used an HTTPRIO component explicitly to be able to monitor the incoming data (which will possibly be very large and slow); for this reason, I set a global `nRead` variable to zero before invoking the remote method. In the `OnReceivingData` event of the HTTPRIO object's `HTTPWebNode` property, you add the data received to the `nRead` variable. The `Read` and `Total` parameters passed to the event refer to the specific block of data sent over a socket, so they are almost useless by themselves to monitor progress:

```
procedure TForm1.HTTPRIO1HTTPWebNode1ReceivingData(
  Read, Total: Integer);
begin
  Inc (nRead, Read);
  StatusBar1.SimpleText := IntToStr (nRead);
  Application.ProcessMessages;
end;
```



any fish

Species No	Category	Common_Name
90020	Triggerfishy	Clown Triggerfish
90030	Snapper	Red Emperor
90050	Wrasse	Giant Maori Wrasse
90070	Angelfish	Blue Angelfish
90080	Cod	Lunartail Rockcod
90090	Scorpionfish	Firefish

1105115

FIGURE 23.7 The FishClient example receives a binary ClientDataSet within a SOAP attachment.

Supporting UDDI

The increasing popularity of XML and SOAP opens the way for business-to-business communication applications to interoperate. XML and SOAP provide a foundation, but they are not enough—standardization in the XML formats, in the communication process, and in the availability of information about a business are all key elements of a real-world solution.

Among the standards proposed to overcome this situation, the most notable are Universal Description, Discovery, and Integration (UDDI, `www.uddi.org`) and Electronic Business using eXtensible Markup Language (ebXML, `www.ebxml.org`). These two solutions partially overlap and partially diverge and are now being further worked on by the OASIS consortium (Organization for the Advancement of Structured Information Standards, `www.oasis-open.org`). I won't get into the problems with business processes; instead I'll only discuss some of the technical elements of UDDI, because it is specifically supported by Delphi 7.

What Is UDDI?

The Universal Description, Discovery, and Integration (UDDI) specification is an effort to create a catalog of web services offered by companies throughout the world. The goal of this initiative is to build an open, global, platform-independent framework to let business entities find each other, define how they interact with the Internet network, and share a global business registry. Of course, the idea is to speed up the adoption of e-commerce, in the form of B2B applications.

UDDI is basically a global business registry. Companies can register themselves on the system, describing their organization and the web services they offer (in UDDI the term *web services* is used in a very wide sense, including e-mail addresses and websites). The information in the UDDI registry for each company is divided into three areas:

White Pages Include contact information, addresses, and the like.

Yellow Pages Register the company in one or more taxonomies, including industrial categories, products sold by the company, geographical information, and other (possibly customizable) taxonomies.

Green Pages List the web services offered by the company. Each service is listed under a service type (called a *tModel*), which can be predefined or a type specifically described by the company (for example, in terms of WSDL).

Technically, the UDDI registry should be perceived like today's DNS, and should have a similar distributed nature: multiple servers, mirrored and caching data. Clients can cache data following given rules.

UDDI defines specific data models for a business entity, a business service, and a binding template. The BusinessEntity type includes core information about the business, such as its name, the category it belongs to, and contact information. It supports the taxonomies of the yellow pages, with industry information, product types, and geographic details.

The BusinessService type includes descriptions of web services (used for the green pages). The main type is only a container for the related services. The services can be bound to a taxonomy (geographical area, product, and so on). Every BusinessService structure includes one or more BindingTemplates (the reference to the service).

The BindingTemplate has a tModel. The tModel includes information about formats, protocols, and security, and references to technical specifications (possibly using the WSDL format). If multiple companies share a tModel, a program can interact with all of them with the same code. A given business program, for example, can offer a tModel for other software programs to interact with it, regardless of the company that has adopted the software.

The UDDI API is based on SOAP. Using SOAP, you can both register data and query a registry. Microsoft also offers a COM-based SDK, and IBM has a Java Open Source toolkit for UDDI. UDDI APIs include inquiry (`find_xx` and `get_xx`) and publishing (`save_xx` and `delete_xx`) on each of the four core data structures (`businessEntity`, `businessService`, `bindingTemplate`, and `tModel`).

UDDI in Delphi 7

Delphi 7 includes a UDDI browser you can use to find a web service while importing a WSDL file. The UDDI browser, shown in Figure 23.8, is activated by the WSDL Import Wizard. This browser uses only UDDI version 1 servers (a newer interface is available, but it isn't supported) and has a few predefined UDDI registries. You can add predefined settings in the `UDDIBrow.ini` file in the Delphi `bin` folder.

This is a handy way to access web services information, but it is not all that Delphi provides. Although the UDDI Browser is not available as a stand-alone application, the UDDI interface units are available (and they are not trivial to import). So, you can write your own UDDI browser.

I'll sketch a simple solution, which is a starting point for a full-blown UDDI browser. The UddiInquiry example, shown in action in Figure 23.9, has a number of features, but not all of them work smoothly (particularly the category search features). The reason lies in the fact that using UDDI implies navigating complex data structures, which are not always mapped in the most obvious way by the WSDL importer. This makes the example code fairly involved; so, I'll show you only the code for a plain search, and not even all of it (another reason is that some readers may not be particularly interested in UDDI).

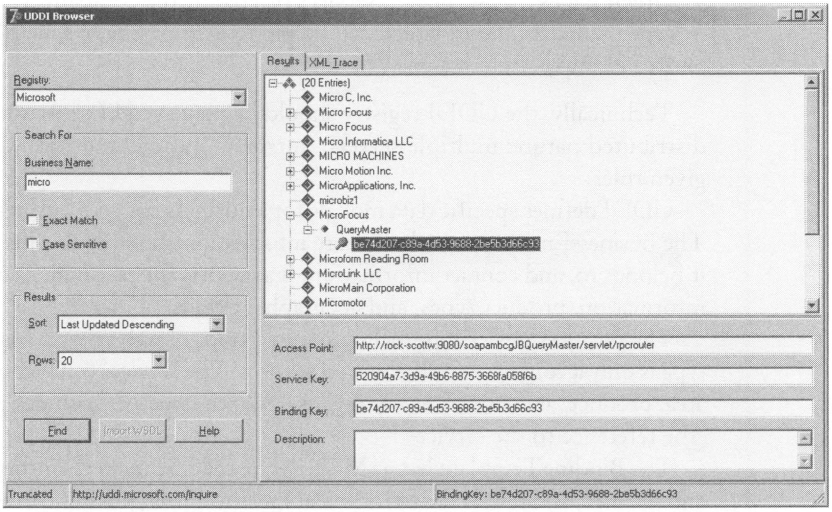


FIGURE 23.8
The UDDI Browser embedded in the Delphi IDE

Registry: Microsoft

Business Name: micro

☐ Exact Match

☐ Case Sensitive

Sort: Last Updated Descending

Rows: 20

- (20 Entries)
  - Micro C, Inc.
  - Micro Focus
  - Micro Focus
  - Micro Informatica LLC
  - MICRO MACHINES
  - Micro Motion Inc.
  - MicroApplications, Inc.
  - microbiz1
  - MicroFocus
    - QueryMaster
      - be74d207-c89a-4d53-9688-2be5b3d66c93
  - Microform Reading Room
  - MicroLink LLC
  - MicroMain Corporation
  - Micromotor

Access Point: http://rock-scottw:9080/soapambcgJBQueryMaster/servlet/rpcrouter

Service Key: 520904a7-3d9a-49b6-8875-3668fa058f6b

Binding Key: be74d207-c89a-4d53-9688-2be5b3d66c93

Description:

Truncated | http://uddi.microsoft.com/inquire | BindingKey: be74d207-c89a-4d53-9688-2be5b3d66c93

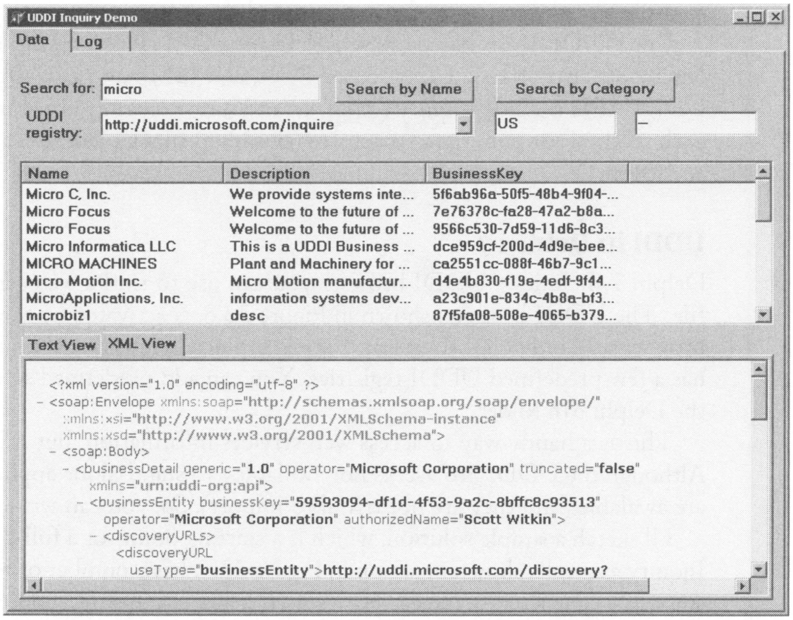


FIGURE 23.9
The UddiInquiry example features a limited UDDI browser.

Search for: micro

UDDI registry: http://uddi.microsoft.com/inquire | US | –

Name	Description	BusinessKey
Micro C, Inc.	We provide systems inte...	5f6ab96a-50f5-48b4-9f04-...
Micro Focus	Welcome to the future of ...	7e76378c-fa28-47a2-b8a...
Micro Focus	Welcome to the future of ...	9566c530-7d59-11d6-8c3...
Micro Informatica LLC	This is a UDDI Business ...	dce959cf-200d-4d9e-bee...
MICRO MACHINES	Plant and Machinery for ...	ca2551cc-088f-46b7-9c1...
Micro Motion Inc.	Micro Motion manufactur...	d4e4b830-f19e-4edf-9f44...
MicroApplications, Inc.	information systems dev...	a23c901e-834c-4b8a-bf3...
microbiz1	desc	87f5fa08-508e-4065-b379...

```
  <?xml version="1.0" encoding="utf-8" ?>
- <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    ::mlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <soap:Body>
    - <businessDetail generic="1.0" operator="Microsoft Corporation" truncated="false"
        xmlns="urn:uddi-org:api">
      - <businessEntity businessKey="59593094-df1d-4f53-9a2c-8bffc8c93513"
          operator="Microsoft Corporation" authorizedName="Scott Witkin">
        - <discoveryURLs>
            <discoveryURL
              useType="businessEntity">http://uddi.microsoft.com/discovery?
```

As the program starts, it binds the HTTPRIO component it hosts with the InquireSoap UDDI interface, defined in the inquire_v1 unit provided with Delphi 7:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  httprio1.Url := comboRegistry.Text;
  inftInquire := httprio1 as InquireSoap;
end;
```

Clicking the Search button makes the program call the `find_business` UDDI API. Because most UDDI functions require many parameters, it has been imported using a single record-based parameter of type `FindBusiness`; it returns a `businessList2` object:

```
procedure TForm1.btnSearchClick(Sender: TObject);
var
  findBusinessData: Findbusiness;
  businessListData: businessList2;
begin
  httprio1.Url := comboRegistry.Text;

  findBusinessData := FindBusiness.Create;
  findBusinessData.name := edSearch.Text;
  findBusinessData.generic := '1.0';
  findBusinessData.maxRows := 20;

  businessListData := inftInquire.find_business(findBusinessData);
  BusinessListToListView (businessListData);
  findBusinessData.Free;
end;
```

The `businessList2` object is a list that is processed by the `businessListToListView` method of the program's main form, showing the most relevant details in a list view component:

```
procedure TForm1.businessListToListView(businessList: businessList2);
var
  i: Integer;
begin
  ListView1.Clear;
  for i := 0 to businessList.businessInfos.Len do
  begin
    with ListView1.Items.Add do
    begin
      Caption := businessList.businessInfos [i].name;
      SubItems.Add (businessList.businessInfos [i].description);
      SubItems.Add (businessList.businessInfos [i].businessKey);
    end;
  end;
end;
```

By double-clicking on a list view item you can further explore its details, although the program shows the resulting XML information in a plain textual format (or a `TWebBrowser`-based XML view) and doesn't further process it. As mentioned, I don't want to get into technical details; if you're interested, you can find them by looking at the source code.

What's Next?

In this chapter, I've focused on web services, covering SOAP, WSDL, and UDDI. Refer to the W3C website and to the UDDI (`www.uddi.org`) and ebXML (`www.ebxml.org`) sites for more information in the area of business-oriented web services. I didn't delve much into these non-technical issues, but they were worth mentioning.

You should have noticed in this chapter that Delphi is strong player in the area of web services, with a powerful, open architecture. You can use web services to interact with applications written for the Microsoft .NET platform; Delphi has much to offer for this architecture, because it includes a Delphi for .NET preview (covered in the two final chapters of the book).

Chapter 24, "The Microsoft .NET Architecture from the Delphi Perspective," is focused on the .NET platform; Chapter 25, "Delphi for .NET Preview: The Language and the RTL," covers the new Delphi compiler.

Chapter 24

The Microsoft .NET Architecture from the Delphi Perspective

EVERY FEW YEARS A new technology comes along that turns our industry upside down. Some of these technologies thrive, some mutate into something else, and many are revealed to be half-baked marketing fluff. Almost in military fashion, the arrival of a new technology is invariably preceded by an artillery barrage of hype. Experienced programmers know to keep their heads down until the hype barrage subsides. After all, the proof will be in the technical pudding, so to speak.

Your reaction to Microsoft's .NET initiative will depend somewhat on your background. If you have previous experience with Java or Delphi, you might wonder what all the fuss is about. But, if you have been slogging it out in the trenches writing applications for Windows in C++ (or, heaven forbid, in C), you might be overwhelmed with joy.

The purpose of this chapter is to explain some of the technologies that make up the .NET initiative and to show how they fit into the world of Windows programming in general and Delphi programming in particular. We will begin by installing and configuring the Delphi for .NET Preview compiler. Then we will broadly cover some of the .NET technologies. Finally, we will discuss these technologies in greater detail, weaving in some Delphi code for illustration.

NOTE *This chapter and the following one were by Marco with extensive help from John Bushakra, who works for Borland's RAD tools documentation department.*

This chapter covers the following:

- Installing the Delphi for .NET Preview
- Microsoft's .NET platform
- Intermediate language
- Managed code and CTS
- Garbage collection
- Versioning

Installing the Delphi for .NET Preview

The Delphi for .NET Preview requires but does not install .NET Framework Runtime, which is freely downloadable from Microsoft's MSDN website. As a developer, you'll probably want to take an extra step and install the more complete .NET Framework SDK, which includes documentation and tools for developers (but is also much larger). You should install the .NET Framework Runtime or SDK before you install the Delphi for .NET Preview.

The Preview compiler is compatible with the .NET Framework SDK and service pack 1. If you have applied service pack 2 or later, then you will have to perform the extra step of rebuilding the precompiled units (the `dcuil` files) installed with the Preview. This requirement might go away in subsequent updates to the Delphi for .NET Preview compiler.

NOTE *A few months after Delphi 7 shipped (in November 2002), Borland released a significant update of the Delphi for .NET Preview. As a registered Delphi user, you can download updates to the Preview from the Borland website. Do this even before installing the version that comes in the Delphi box, as you'll need to uninstall it to update to a newer version.*

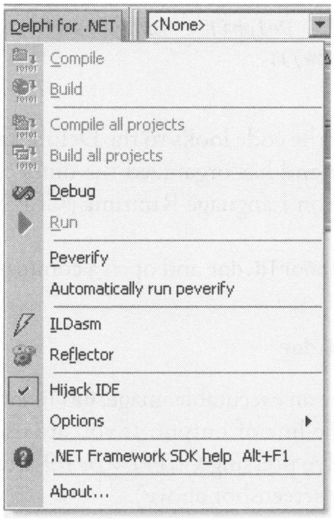
WARNING *I've used the November 2002 version of the Delphi for .NET Preview to test the examples of this and the .NET chapter, although most of them will also compile against the original version that shipped with Delphi 7. By the time you read this, a newer version might be available; check the author's website for updates of the examples.*

After installing the .NET Framework SDK, insert the Delphi for .NET Preview CD and run the setup program. The Preview will install itself in a separate directory from your Delphi 7 installation, and none of your Delphi 7 settings are affected by installing the Preview.

Previously I mentioned that you would need to rebuild the precompiled units shipped with the compiler, if you have installed service pack 2 for the .NET Framework. If this applies to you, open a command window and navigate to the `source\rtl` directory under the root of the compiler installation directory. There, you will find a file called `rebuild.bat`, which you must execute to perform the rebuild. You might see some errors and warnings; these are known issues at the time of this writing, and might be cleared up in subsequent updates to the compiler. Once the batch file has completed, you will be ready to go.

The dccil compiler is located in the `bin` directory. Along with the compiler, in this directory is a file called `dccil.cfg` that specifies the default unit search path (the `-U` compiler switch), which is the `units` directory under the root installation directory. As packaged, the Delphi for .NET Preview compiler is strictly for use with a command window.

However, Borland has made available an unsupported plug-in to the Delphi 7 IDE on the Borland Developers Network website, referred to as Delphi for .NET common line compiler IDE integration. Using the plug-in, you can control the dccil compiler from within the IDE, as you can see in the menu installed by this plug-in.


Delphi for .NET
<None>
Compile
Build
Compile all projects
Build all projects
Debug
Run
Peverify
Automatically run peverify
ILDasm
Reflector
Hijack IDE
Options
.NET Framework SDK help Alt+F1
About...

Note, however, that the plug-in does not give you the full capability of designing forms and all the other things Delphi is famous for. The Delphi for .NET Preview compiler is intended to give you some advance warning about new language features and a glimpse into how the Delphi run-time library might look in the .NET context. You can download this plug-in from the Code Central area of the Borland Developer Network website, `bdn.borland.com` (if you cannot find it, look under the number 18889).

Another valuable resource you might want to work with is the Reflector. This tool was written by Lutz Roeder (who happens to work at Microsoft, by the way) and is available on his own website, `www.aisto.com/roeder/dotnet`.

NOTE *Reflector is like Microsoft's Intermediate Language Disassembler (ILDASM) tool—it allows you to inspect .NET assemblies (executables and DLLs) and their types and members.*

Testing the Installation

Now it's time to test your Delphi for .NET Preview installation by compiling a simple console program that prints a message.

You can start this project using Delphi 7, or you can type in the text with your favorite editor:

```
program HelloWorld;

{$APPTYPE CONSOLE}

uses
  Borland.Delphi.SysUtils;
```

```
begin
  WriteLn ('Hello, Delphi! - Today is ' +
    DateToStr (Now));
end.
```

Notice how similar the code looks to the Delphi applications you're used to. The only difference is the `uses` clause. Borland has organized the units of the Delphi library into namespaces similar to those of the Common Language Runtime (CLR). I will discuss this in more detail in the next chapter.

Save the file as `HelloWorld.dpr` and open a command window. Navigate to the location of the file and type

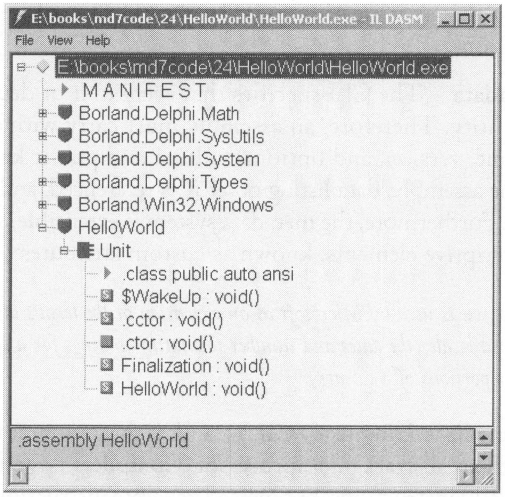
```
dccil HelloWorld.dpr
```

The result should be an executable image, `HelloWorld.exe`, which you can run from the command line to produce a single line of output. If you are using the IDE plug-in, compiling and running the program will involve pressing Ctrl+F9 or F9 after you've enabled IDE *hijacking* (see the Hijack IDE menu item in the screenshot above).

If this program seems boring, you can begin your exploration of .NET using the windowed version of the hello world program instead of the console version. Called HelloWin, this program creates and shows a window on the screen, writing into its caption. It also shows the call to `Application.Run`, used to activate the application's message processing loop:

```
program HelloWin;

uses
  System.Windows.Forms,
  Borland.Delphi.SysUtils;

var
  aForm: Form;

begin
  aForm := Form.Create;
  aForm.Text := 'Hello Delphi';
  Application.Run (aForm);
end.
```

In .NET presentations, the previous code raises the enthusiasm of programmers used to Microsoft development tools. As a Delphi programmer, you might have used almost this same code since 1995, so you might wonder where this excitement comes from. The output of these programs is far from interesting, but you can at least prove they are not classic Win32 applications by running ILDASM on them (directly from the plug-in's menu). You can see the output for the HelloWorld executable in Figure 24.1. Notice that the unit's global code is wrapped in a pseudo class called `uUnit`. Unlike Delphi, the .NET runtime doesn't account for global procedures and functions, but only class methods. But I'm getting ahead of myself; let's start from the beginning and look at the NET platform.



```
E:\books\md7code\24\HelloWorld\HelloWorld.exe
  MANIFEST
  Borland.Delphi.Math
  Borland.Delphi.SysUtils
  Borland.Delphi.System
  Borland.Delphi.Types
  Borland.Win32.Windows
  HelloWorld
    Unit
      .class public auto ansi
      $WakeUp : void()
      .cctor : void()
      .ctor : void()
      Finalization : void()
      HelloWorld : void()
.assembly HelloWorld
```

FIGURE 24.1
The HelloWorld demo as seen in ILDASM

Microsoft's .NET Platform

Microsoft's .NET platform comprises many different specifications and initiatives. The core functionality of the .NET platform has been given over to the European Computer Manufacturer's Association (ECMA) and is undergoing the standardization process. At the time of this writing, the C# language specification has just passed ECMA's process and is heading for ISO standardization.

NOTE *You can find the standard documents for the C# programming language and the Common Language Infrastructure (CLI) at* `www.ecma.ch`*. An interesting element of the CLI specification is the standard naming convention for variables and methods.*

From the programmer's standpoint, the core feature of the .NET platform is its managed environment, which allows the execution of intermediate code compiled by many different programming languages (provided they conform to a common definition of the base data types). This environment embeds many features ranging from sophisticated memory management up to integrated security, to name just two. On top of this managed environment, Microsoft has built a large class library, covering diverse areas of development (Windows-based forms, web development, web services development, XML processing, database access, and many more).

This is only a short overview. To get more into the details we need to learn the precise terms used in the .NET Platform, most of which are indicated by of three-letter acronyms, introduced in the following subsections.

The Common Language Infrastructure (CLI)

The CLI is the main pillar of the .NET platform. It was submitted to the ECMA in December 2001. The CLI comprises many different specifications:

Common Type System (CTS) The CTS lays the groundwork for integrating disparate programming languages. It specifies how types are declared and used, and language compilers must conform to this specification in order to take advantage of the .NET platform's cross-language integration.

Extensible Metadata The CLI specifies that every unit of deployment (an *assembly*) must be a self-describing entity. Therefore, an assembly must carry with it data that fully identifies the assembly (its name, version, and optional culture and public key), data that describes the types defined within the assembly, data listing other files referenced, and any special security permissions that are required. Furthermore, the metadata system is extensible, so an assembly might also contain user-defined descriptive elements, known as custom attributes.

NOTE *The term* culture *is used by Microsoft as an extension of the term's language (the language used for the user messages of a library) and locale (the dates and number formatting settings for a country). The idea is to cover anything peculiar to a country, or portions of a country.*

Common Intermediate Language (CIL) CIL is the programming language of a virtual execution environment with an abstract microprocessor. Compilers targeting the .NET platform do not generate native CPU instructions, but instead generate instructions in the intermediate language of the abstract processor. In this respect, the CLI is similar to Java byte code.

P/Invoke Programs that execute in the .NET runtime each play in their own private sandbox. Unlike the traditional Win32 programming model, a large and complicated layer of software exists between them and the operating system. But the .NET runtime does not completely replace the Win32 API, so there must be a way to bridge between the two worlds. This bridge is called Platform Invocation Service (in short, P/Invoke or PInvoke).

Framework Class Library (FCL) The .NET Framework Class Library (FCL), or .NET Framework for short (or even .NET Fx), is a class hierarchy with a design similar to that of Borland's VCL. Features also available in the VCL are actually quite similar, although the .NET Framework has a much larger set of classes (covering many more areas of programming). The architectural difference between the .NET Framework and the VCL is that the .NET Framework is not only callable from other languages, it can be directly extended by other languages. This means that, as a Delphi programmer, you have the ability to directly inherit from classes in the .NET Framework, just as you would inherit from any other Delphi class. Moreover, the CLI gives you the ability to extend a class written in any language that has a compiler that targets the .NET runtime. The *factorable* part of the FCL means that parts of the class hierarchy can be factored out; for example, to create a stripped-down version for use on a handheld device.

Extended Portable Executable (PE) File Format Microsoft is using its standard Portable Executable (PE) file format (the file format used by standard Win32 executable files) to support the metadata requirements of the CLI. The advantage of using this format is that the operating system can load a .NET application the same way it loads a native Win32 application. Common loading is about where the similarity ends though, since all managed code is in a special section. The framework modifies the loader so when it finds it is dealing with a .NET entity, it passes control over to the CLR that then works out how to call the managed entry points.

The Common Language Runtime (CLR)

The CLI is a specification, and the CLR is Microsoft's implementation of that specification. Not surprisingly, the CLR is a superset of the specification. To a programmer, the CLR is an all-encompassing run-time library that unifies the vast array of services provided by the Windows operating system and presents them to you in an object-oriented framework.

On a larger scale, the CLR is responsible for every aspect of the .NET universe: loading the executable image, verifying its identity and type safety, compiling CIL code into native CPU instructions, and managing the application during its lifetime. CIL code that is intended to be run in the CLR is called *managed code*, whereas all other code (such as Intel executable code produced by Delphi 7) is *unmanaged*.

COMMON LANGUAGE SPECIFICATION (CLS)

Closely related to the Common Type System, the CLS is a subset of that specification that defines the rules that govern how types created in different programming languages can interoperate. Not all languages are created equal, and some languages have features that can't be found elsewhere. The CLS tries to find a happy medium, specifying those items that a large set of languages can implement. Microsoft tried to make the CLS as small as possible, yet still accommodate a large set of languages.

Some features of Delphi are not CLS compliant. This does not mean the code cannot be executed by the CLR and work on the .NET platform; it only means you are using a language feature that can't be supported by other languages, so that particular part of your code cannot be used by other .NET applications if they are written in languages other than Delphi.

NOTE *If you have experience with Java, you might find some of these items familiar. The .NET platform shares many concepts with the Java platform, in particular the intermediate language and virtual execution system, although with the core difference of Java being pcode interpreted by default, whereas .NET is invariably JIT compiled. There are also relevant analogies in the class libraries. In most cases, however, don't take corresponding Java concepts for granted, because differences in the implementation details might significantly affect how you use an apparently similar feature.*

The various specifications of the CLI give a glimmer of hope about cross-platform development. But, you probably won't hear the "write once, run anywhere," mantra from Microsoft. This is due to the fact that they consider the user interface to be a key part of an application, and a normal PC screen compared to, say, a mobile phone screen has inherently different capabilities. There are two major ongoing efforts to implement the CLI on other operating systems. The Rotor project (officially called Microsoft Shared Source CLI or MS SSCLI) is written by the Microsoft Research team. Rotor consists of a large donation of software, under Microsoft's Shared Source license, and the necessary tools to build it on the FreeBSD, Win2, and Mac OS X 10.2 operating systems.

The second well-known CLI implementation on another operating system is the Mono Project, which supports Win32 and Linux. Mono is building what is essentially a clean-room implementation of the CLI on Linux and is released with a much more open license. Borland has a wait-and-see approach about Mono; there has been no word about whether Delphi will be a player in that arena.

The Rotor project seems to be aimed toward educators and people who are just curious about how the CLR is implemented, because its license is very open to academia but prohibits any commercial use. The Mono project, however, may give Microsoft some serious competition. The most serious stumbling block to true portability will probably continue to be the graphical user interface. The Rotor project does not include any GUI elements, while Mono has started developing WinForms using the WINE library. There is also a GTK# project associated with Mono, which is a binding of the GTK+ toolkit to the C# language.

Today's computing environment is a nebulous mass of diverse possibilities. Handheld devices have complex operating systems of their own, and Microsoft has a keen interest in this arena. Microsoft is working on a version of the .NET platform called the .NET Compact Framework, which is intended for handheld devices. The .NET platform could also serve as a springboard for tomorrow's technologies. Sixty-four bit processors are just around the corner, and .NET will no doubt be running there.

Does this mean you can run your managed application on Linux, 64-bit Windows, and your PDA? No. It is not reasonable to expect a user interface geared for a 1600×1200 pixel display on a 21-inch monitor to port to a handheld device. So, regarding .NET as a cross-platform tool, you will gain some advantage, but you should not expect your application to be a straight port especially if you are porting to another operating system or a handheld device.

In the following sections, we will examine some components of the CLI in more detail.

Assemblies

Microsoft coined the term *assembly* to denote a "unit of deployment" in the .NET runtime environment. In manufacturing, an assembly is a group of separate but related parts that are put together to form a single functional unit. Sometimes an assembly consists of only one part, but it might consist of many. So it is with assemblies in .NET.

Typically, an assembly consists of only one file—either a single executable or a dynamic link library. An assembly could consist of a group of related DLLs, or a DLL and associated resources like images or strings. Going back to the manufacturing analogy, each assembly, whether it consists of one part or multiple parts, contains a packing slip called a *manifest*. A manifest describes the contents of something, and an assembly manifest contains the metadata for an assembly. The manifest is the implementation of the extensible metadata system described in the CLI.

As you look through the installation directory for the Delphi for .NET Preview compiler, you will find a `units` folder. In this folder are a number of files with the extensions `.dcua` and `.dcuil`. These are not PE files, so they cannot be examined with ILDASM.

A `.dcua` file catalogs the namespaces and types in a .NET assembly (which, keep in mind, can consist of multiple executable files). A `.dcuil` file corresponds to a namespace. The `.dcuil` file contains all the compiler symbols for a namespace, as well as references to the `.dcua` files that contribute to that namespace (an assembly can contribute types to more than one namespace).

NOTE *A* namespace *is a hierarchical containment mechanism that organizes classes and other types. We will discuss namespaces in more detail in Chapter 25, "Delphi for .NET Preview: The Language and the RTL."*

An application is built against a certain set of assemblies; the application is said to *reference* these assemblies. When the set of assemblies changes, the compiler must rebuild any `.dcuil` files that contain references to `.dcua` files that are no longer in the set. Likewise, if a change is made to an assembly, the compiler must rebuild the corresponding `.dcua` and `.dcuil` files for that assembly. A `.dcuil` file is roughly equivalent to a Delphi `.dcu` file, but a `.dcua` file does not have a corresponding file type in the Delphi for Win32 universe.

Different project types produce different types of .NET assemblies: A Delphi `program` produces an executable assembly; a `library` project produces a DLL assembly.

The Intermediate Language

The Common Language Runtime is an implementation of a virtual execution system, or virtual machine. Like all virtual machines, the CLR has its own abstract microprocessor. As already mentioned, the assembly language of the virtual processor is called Common Intermediate Language (CIL), although before being promoted as a standard it was called Microsoft Intermediate Language (MSIL), a terms you'll still see around often. Compilers that target the CLR do not generate code in the native instruction set of any specific, real microprocessor. Instead, .NET compilers target the abstract processor of the CLR. The hardware abstraction built into the CLR hints at some cross-platform viability. Remember, Microsoft's CLR is but one implementation of the CLI; any hardware/operating system platform that has a CLI-compliant execution environment built for it could be targeted.

Of course, there is no real microprocessor that can execute CIL directly, so it must be compiled to the instruction set of the native hardware prior to execution. This is the job of the Just in Time (JIT) compiler. Here is where the CLR differs from other virtual machine implementations (like Java). The CLR is not an interpreter, nor does it execute bytecode. On the .NET platform, CIL is always compiled to native CPU instructions, and once compiled it is cached in memory; so chances are good that it will never have to be recompiled.

NOTE *In some memory-constrained environments (such as a PDA), compiled code can be discarded. In this case, the code would need to be recompiled if it was ever reloaded.*

Compiling IL is not a very expensive operation (MS Research has spent years developing technology to allow the JIT compilation to be as negligible as possible) but does imply a little overhead and it must be repeated every time you run even the same program. Most applications will see a little increase in startup time (what's particularly slow is loading all the .NET framework itself with the first .NET application run in a Windows session); however, this is limited because not all code in an application is compiled at once. The JIT compiler works in conjunction with the loader, and so IL code is not compiled until it is called (on a method by method basis).

JIT compilation is the normal case on the .NET platform, but it is possible to compile a managed executable into native instructions and store the native image on disk. By doing so, you avoid the negative impact of JIT compilation on your application's startup time. The .NET Framework runtime contains a utility called the Native Image Generator (`Ngen.exe`) to accomplish this.

The native image created by Ngen is stored in the native image cache. The next time the CLR tries to load the assembly, it looks in the native image cache first. If a native image of the assembly is found, it is preferred over the IL version. Note that you must also deploy the IL version of the assembly, because the native image does not contain any metadata. In addition, the end user or administrator could remove your native image from the cache. In this case, there would be no native image to find, and the CLR would revert back to the usual JIT compilation of the IL version.

The ability to generate a native executable image can be helpful, but you should profile your application under both environments (JIT and native image) to see whether the detrimental effects of the JIT compiler are that bad. The JIT compiler is, after all, a real compiler for a specific microprocessor, and as such it can do some performance optimizations of its own. It employs good algorithms to reduce its overhead and also introduces optimizations to the compiled code (somewhat like Delphi does in its compilations).

Looking over the IL code generated by your compiler can be highly educational. Microsoft provides a utility called the IL Disassembler (`ildasm.exe`) that you can use to dissect your assemblies at the lowest level. Located in the `bin` directory of your .NET Framework SDK installation, ildasm can load any assembly and its metadata: the manifest, the classes, their methods and properties, and, of course, the IL code generated by the compiler. We will look more at ILDASM in this chapter and the next. The Reflector tool mentioned earlier is also useful in this regard.

Managed and Safe Code

Simply put, *managed code* is any code that is loaded, JIT compiled, and executed under the auspice of the CLR. Like all executable and library modules on the Windows platform, managed code modules are stored in Microsoft's Portable Executable (PE) file format. A managed PE file contains additional header information and, when loaded, jumps into the runtime's execution engine (to a function in `MSCorEE.dll`).

The runtime initializes, and then it looks for the module's entry point. The IL code in the entry point is JIT compiled to native CPU instructions. Finally, execution begins at the module's entry point. The situation is similar for a library module; the PE file directs the loader to jump to a different function in `MSCorEE.dll`.

In contrast, *unmanaged code* consists not of IL, but rather of traditional, native CPU instructions. Unmanaged code executes outside of the runtime and therefore can't take advantage of the services provided by the CLR—at least not without special measures.

Unmanaged code can create .NET Framework classes using COM Interop services. The .NET Framework class is wrapped in a COM proxy and exposed to unmanaged code as if it were a COM object. The COM Interop bridge goes the other way too, allowing a COM objects in a COM server to be accessed by managed code. Finally, the Platform Invoke services of the CLR allow managed code to call the Win32 API directly.

NOTE *The Delphi for .NET Preview compiler produces fully managed code. There is currently no support for mixing managed and unmanaged (native) code within the same module, as you can with Microsoft's Visual C++ .NET (which uses a mechanism called IJW: It Just Works).*

A module is completely self-describing, because it contains both IL code and metadata that describes the data elements used by the code. Taking the IL code and the metadata together, the CLR can perform another level of verification beyond the static checking done by the compiler. This process, which is always performed unless a system administrator turns it off, verifies that code is type safe. Verifiably type-safe code is known as *safe code*. Safe code passes the following type-safety checks:

- Only valid operations are invoked on objects. This includes parameter validation, return type verification, and visibility checks.

- Objects are always assigned to compatible types.
- The code uses no explicit pointers, as they might refer to invalid memory locations.

As you'd expect, *unsafe code* fails to pass these checks. But, just because code is not verifiably type safe does not mean it is unsafe; it simply means the code could not be verified, either due to a limitation in the verification process, or perhaps in the compiler itself. When it will be released as a finished product, the Delphi for .NET compiler is expected to generate verifiably type-safe code.

Some Delphi language constructs are not CLS compliant, but this is different than not being verifiably type safe. Non-CLS-compliant language constructs will be covered more in Chapter 25.

The .NET Framework SDK contains a PEVerify utility that exhaustively analyzes a managed PE for type safety (`peverify.exe`). The Delphi 7 IDE plug-in mentioned previously lets you automatically run PEVerify on your code after every build.

The Common Type System

The Common Type System (CTS) is the bulldozer that levels the playing field for programming languages in the .NET framework. The CTS fully specifies the primitive types and object types known to the CLR. These types are used to define an object model that is shared among all languages that target the CLR.

The Component Object Model (COM) has been the usual way to achieve binary compatibility and language interoperability on the Windows platform. The CTS goes beyond that, allowing languages as different as Eiffel, C#, and Delphi to integrate with each other. Components written in these disparate languages can pass objects among themselves and directly extend their capabilities through inheritance. This level of integration of programming languages is unprecedented.

All types defined by the CTS fall into two categories: value types and reference types. Value types, as their name implies, have pass-by-value semantics. For example, say you have a variable that is a value type. If you pass this variable as a parameter to a function and modify the parameter within the function, the original variable will be unaffected. Examples of value types include scalar types, enumerations, and records. Aggregate types such as Delphi records (or C# structures) are known as value classes within the CTS.

On the other hand, reference types have alias, or pass-by-reference, semantics. If you have a variable that is a reference type (for example, an instance of a class) and you pass that variable as a parameter to a function, any changes you make to the parameter will also affect the variable. Examples of reference types include class types and interfaces. Pointer types are also reference types, as are delegates, which will be discussed shortly.

OBJECTS AND PROPERTIES

Like Delphi, the CTS implements a single-inheritance model. A class must inherit from one and only one ancestor and may declare itself to be an implementer of zero or more interfaces. Other familiar object-oriented attributes of the CTS are private, public, and protected visibility of classes and class members (with other visibility specifiers available, as discussed in the next chapter). These CTS visibility specifiers have meanings similar to those in Delphi; however, they are more restricted, in line with C++ semantics. (In Chapter 25 we'll look at how Delphi's visibility specifiers map to the CTS versions, and examine the specific changes made to the language thus far to accommodate additional features of the CTS.)

As you peruse the .NET literature, you will notice similarities between the capabilities of CTS class types and Delphi classes. The traditional object-oriented features of fields and methods are supported, of course. In addition, the CTS implements properties in a way that is conceptually similar to the familiar Delphi notion. Properties in CTS can have read and write access methods that restrict or compute values on the fly, or they can simply mask private fields. However there are also many differences, including the fact that property get/set methods must have the same visibility as the property itself to ensure languages that don't support property syntax can still access the property. Although Delphi doesn't enforce this in the source code, it modifies compiled code behind the scenes if necessary.

EVENTS AND DELEGATES

One reason the Win32 API has survived so long is that at its lowest levels it is based on fundamental concepts, such as using the address of a function as a callback mechanism. The entire Windows user interface event system is based on callback functions (and some of events in the VCL framework are built on top of even that system). The callback mechanism is so powerful that it surely must make its way into the CTS. Using callbacks in a type-safe, language-neutral way relies on a reference type called a *delegate*.

CTS delegates are different from ordinary function pointers, in that they can reference both static and instance methods of a class. The declaration of a delegate must match the signature of the methods the delegate will reference. In Delphi for .NET, usage of delegates is similar to that of familiar procedural types:

```
type
  TMyClass = class
  public
    procedure myMethod;
  end;

var
  threadDelegate: System.Threading.ThreadStart;
  tmc: TMyClass;
  aThread: System.Threading.Thread;

begin
  tmc := TMyClass.Create;
  threadDelegate := @tmc.myMethod;
  aThread := Thread.Create (threadDelegate);
  aThread.Start;
```

The `threadDelegate` variable is of type `System.Threading.ThreadStart`, which is a CLR delegate class. Methods you assign to a delegate have a signature matching the delegate's, which in this case is a procedure taking no parameters. (You can find this code snippet in the `Delegate` sample folder.)

The compiler is hiding a lot of complexity here. Behind the scenes, the compiler must create an instance of the class `System.MulticastDelegate`. The delegate—the function being encapsulated (`myMethod` in this case)—is invoked using methods on the `MulticastDelegate` class. This class supports the simultaneous encapsulation of multiple functions in a single delegate. In a user interface event model, this translates to having multiple listeners for an event.

NOTE *Incidentally, the fact that delegates are classes tells you why you can declare a delegate outside the scope of a class. Because delegates are instances of* `System.MulticastDelegate` *(or a compiler-generated descendant class), you can declare them anywhere you can declare a class.*

Each specific language compiler implements some form of semantic sugar to make the creation of events and the addition and removal of event listeners less painful. For example, in C#, Microsoft used the `+=` and `-=` operators to add and remove functions from the underlying delegate. Delphi for .NET uses set semantics for the same purpose. In Chapter 25 we'll explore this topic, showing how the functions `Include` and `Exclude` are used to assign event handlers. We'll also look at how Delphi's `:=` assignment operator works, with regard to assigning event handlers in the .NET universe.

Garbage Collection

The garbage collector is that part of the CLR that performs automatic management of memory allocation and deallocation. All Delphi 7 offers in this respect is reference counting for interface-based variables (see Chapter 2 for details). When there are no more references to an object, the memory for it is reclaimed. This is also the basic idea of the CLR garbage collector, but the similarities stop there. The CLR's garbage collector can detect that two objects are still referring one to the other, but there is no other reference to them, so they can be discarded, something a reference counting system (like the one in Delphi) fails to handle.

Using a GC means you can create objects, reference one from another as you need, and simply forget about the hassles of deleting those objects. The system will do all that's needed for you. That's all you really need to know. Note that this implies that you don't need to *balance* object creation and destruction in your code architecture, nor do you need to use `try/finally` blocks to make sure objects are destroyed; these are just a couple of specific circumstances in which you'll benefit from a GC.

Only when you need to write low-level classes will you have to remember to free unmanaged resources, such as window or file handles. The CLR class `System.Object` contains a protected `Finalize` method that you can override to free up unmanaged resources. You must be aware of some important things before you begin overriding `Finalize`. (If you want some in-depth information on this topic, see the sidebar "Issues with Overriding Finalize.")

ISSUES WITH OVERRIDING FINALIZE

Overriding the `Finalize` method is inefficient because it requires your object to make at least two trips through the garbage collector. The job of the garbage collector is to reclaim memory, and it can't do this until it runs your object's `Finalize` method. So, the garbage collector must give special treatment to all objects with `Finalize` methods. It puts them on a special list, which has the side effect of keeping them alive through this round of garbage collection (because there is now another reference to the object). Eventually, a thread walks through the special list, runs each object's `Finalize` method, and removes the object from the list. Removing the object from the list takes away the last reference to it, so next time the garbage collector runs, the object's memory can be reclaimed.

Continued on next page

ISSUES WITH OVERRIDING FINALIZE *(continued)*

But inefficiency is not the only problem with finalizers. You also can never be sure when your object's finalizer will run. If you are relying on the `Finalize` method to give back unmanaged resources, the object might hold onto these resources longer than you expect.

When `Finalize` runs, it does not run in the same thread of execution as your object. This means all thread synchronization and blocking must be avoided, because it's not your thread you are blocking, it's the CLR's finalizer thread. On a related note, if an exception is thrown from `Finalize` the CLR catches it, not you, and the exception will be swallowed.

The recommended approach for freeing up unmanaged resources is to implement the dispose pattern. This is a strategy for providing an object both an automatic way to dispose the external resources it refers to when it is garbage collected and also a way to free the same resources upon the direct invocation of a method. In real terms, coding this pattern equates to implementing an interface called `IDisposable`. `IDisposable` consists of one method: `Dispose`. The `IDisposable` interface gives you deterministic control over freeing up resources used in your objects. Unlike `Finalize`, the `Dispose` method is public and is meant to be called by you (or users of your class), not by the garbage collector.

You may have read .NET literature explaining how the C# language implements destructor semantics by having the compiler create a `Finalize` method on your class behind the scenes. To implement the `IDisposable` interface in C#, you must do so directly and wire everything so that the `Dispose` method can be called directly or from the automatically generated `Finalize` method. This is not the way things work in Delphi for .NET.

Creating a destructor for your class does not cause the compiler to implement a finalizer behind the scenes. Instead, it causes the compiler to implement the `IDisposable` interface. But nothing is stopping you from implementing `IDisposable` directly, so if you want to rely on the compiler's behind-the-scenes magic, you must follow a strict pattern. Your class destructor must be declared exactly as shown here:

```
destructor Destroy; override;
```

When the compiler sees this declaration, it generates code to mark your class as an implementor of `IDisposable`. Then, using a feature of CIL, your destructor is marked as the implementation of the `Dispose` method (this can be done even though the name of the method is `Destroy`, not `Dispose`).

The usual way to dispose of objects in Delphi is to call the `Free` method on the object. In Delphi for .NET, the `Free` method is implemented so that it tests to see whether the object implements the `IDisposable` interface. If you follow the previous destructor signature, the compiler implements it for you; `Free` then calls `Dispose`, which winds up in your `Destroy` method.

Let's create a project and declare a class with a destructor that follows the pattern. Then you'll use ILDASM to inspect the generated code. The code is available in Listings 24.1 and 24.2.

LISTING 24.1: THE PROJECT OF THE DESTRUCTORTEST EXAMPLE

```
program DestructorTest;

{$APPTYPE CONSOLE}
```

```
uses
  MyClass in 'MyClass.pas';

var
  test : TMyClass;

begin
  WriteLn ('DestructorTest starts');
  test := TMyClass.Create;
  test.Free;
  WriteLn ('DestructorTest ends');
end.
```

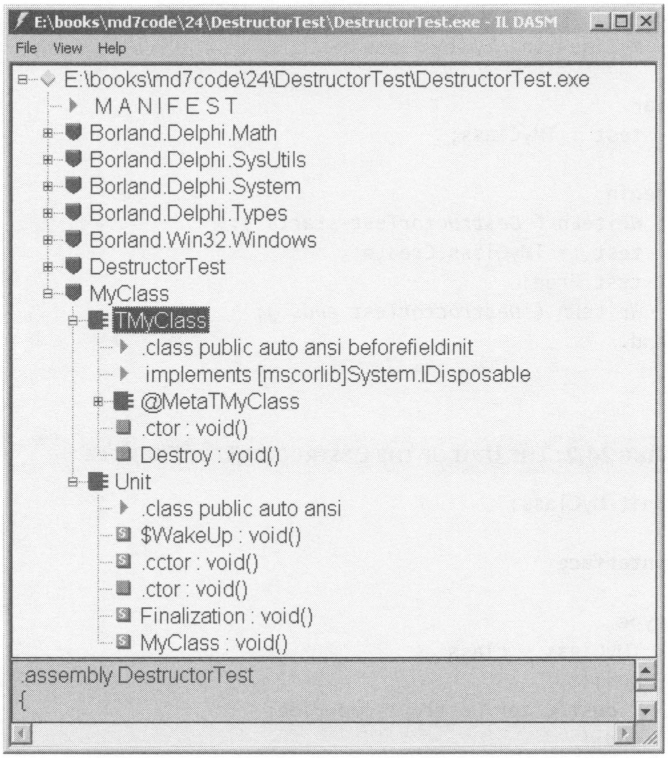
LISTING 24.2: THE UNIT OF THE DESTRUCTORTEST EXAMPLE

```
unit MyClass;

interface

type
  TMyClass = class
  public
    destructor Destroy; override;
  end;

implementation

destructor TMyClass.Destroy;
begin
  WriteLn ('In destructor (which is actually the IDisposable.Dispose method)');
end;

end.
```

After compiling this program you can run it; but we are not so much interested in running the program as in inspecting it with ILDASM. Start ILDASM and select File | Open. Navigate to the directory where `DestructorTest.exe` is located, and open it. You will see a window similar to that shown in Figure 24.2.

The tree entry labeled `MyClass` represents the namespace created to hold all the symbols in the unit. Expand the `MyClass` node and notice the entries for `TMyClass` and `Unit`. Delphi for .NET creates a CLR class for each unit to implement initialization and finalization and also to let you still write global routines; they become methods of that `Unit` class. Expand the node for `TMyClass` and notice the `Destroy` node, which has a pink block. This node represents the `Destroy` method in `TMyClass`. Double-click the `Destroy` node to open a window displaying the full IL code for this method, as shown in Figure 24.3.

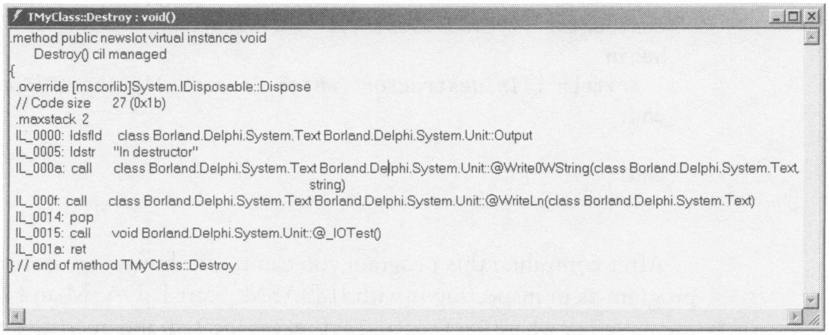
FIGURE 24.2
The ILDASM output for the DestructorTest example


E:\books\md7code\24\DestructorTest\DestructorTest.exe - IL DASM
File View Help

```
E:\books\md7code\24\DestructorTest\DestructorTest.exe
    MANIFEST
    Borland.Delphi.Math
    Borland.Delphi.SysUtils
    Borland.Delphi.System
    Borland.Delphi.Types
    Borland.Win32.Windows
    DestructorTest
    MyClass
        TMyClass
            .class public auto ansi beforefieldinit
            implements [mscorlib]System.IDisposable
            @MetaTMyClass
            .ctor : void()
            Destroy : void()
        Unit
            .class public auto ansi
            $WakeUp : void()
            .cctor : void()
            .ctor : void()
            Finalization : void()
            MyClass : void()
```

```
.assembly DestructorTest
{
```

FIGURE 24.3
The ILDASM IL window for the `Destroy` destructor



```
.method public newslot virtual instance void
    Destroy() cil managed
{
 .override [mscorlib]System.IDisposable::Dispose
 // Code size    27 (0x1b)
 .maxstack 2
 IL_0000: ldsfld   class Borland.Delphi.System.Text Borland.Delphi.System.Unit::Output
 IL_0005: ldstr    "In destructor"
 IL_000a: call     class Borland.Delphi.System.Text Borland.Delphi.System.Unit::@Write0WString(class Borland.Delphi.System.Text,
                   string)
 IL_000f: call     class Borland.Delphi.System.Text Borland.Delphi.System.Unit::@WriteLn(class Borland.Delphi.System.Text)
 IL_0014: pop
 IL_0015: call     void Borland.Delphi.System.Unit::@_IOTest()
 IL_001a: ret
} // end of method TMyClass::Destroy
```

The line you're looking for uses the `.override` directive:

```
.override [mscorlib]System.IDisposable::Dispose
```

This is an explicit override, and it says that the `Destroy` method is the implementation of `Dispose` in the `IDisposable` interface. The `.override` directive is how the thread of execution winds up in your destructor when `Dispose` is called from `Free`.

NOTE *The way* `IDisposable` *was dealt with illustrates a familiar pattern in the ongoing work on the Delphi for .NET compiler. Unlike C#, Delphi is an old language with a large and loyal following and a sizable existing code base. Borland could not sweep everything aside and mandate a new way of doing things that would break all existing code. The implementation of* `IDisposable` *was done in a way that should allow Delphi programmers to continue using familiar paradigms and minimize the impact of porting their existing code to the .NET platform.*

Garbage Collection and Efficiency

Among programmers, the use of a garbage collector (GC) is the topic of many debates. Most programmers like the fact that a GC helps them reduce the chance of memory management errors. Some programmers, though, fear that a GC might not be efficient enough, a fear that often prevented the widespread use of this technology. The inefficient GC of the early Java virtual machines didn't really help in this respect. Even Microsoft has a hard time convincing all programmers to use its GC, up to the point that they overrate their solution depicting it as perfect. Even in Microsoft technical documentation about the GC, you'll find lots of hype and little facts.

NOTE *This is not to say that the GC doesn't do an adequate job; quite the contrary. But it works differently than it is presented. At present, your only way to know how the GC works is to write test case programs and study their effect.*

As a starting point for your exploration, you can use the GarbageTest example. Use a Windows memory analysis tool to see how the memory is affected by program execution. The GarbageTest example declares the following class of large objects (about 10 KB):

```
type
  TMyClass = class
  private
    data: Integer;
    list: array [1..10240] of Char;
    s: string;
  public
    constructor Create;
  end;
```

The simplest test code is the following:

```
for i := 1 to 10000 do
begin
  mc := TMyClass.Create;
  WriteLn (mc.s);
end;
```

Writing this simple program in Delphi will flood the memory, because there is no call to `Free`. In .NET, however, the GC reuses the single memory block, so memory consumption is flat. If you save each object in an array declared as

```
objlist: array [1..10000] of TMyClass;
```

the memory consumption will increase at each cycle of the loop, causing problems! Keeping the object references in memory and then releasing them regularly or randomly can provide more complex tests. You'll find a few snippets in the program code, but you'll have to exercise your knowledge and imagination to build other significant test cases.

Deployment and Versioning

Most Windows software developers have at some point felt the pain caused by deploying shared libraries. The .NET Framework can adapt to a number of different deployment scenarios. In the simplest scenario, the end user can copy files to a directory and go. When it's time to remove the application from the machine, the end user can remove the files, or the entire directory, if it doesn't contain any data. In this scenario there is no installer, no uninstaller, and no registry to deal with.

TIP *You can package your application for the Microsoft Installer if you want to and still deploy to a single directory, with no Registry impact.*

Outside of the easy scenario, the deployment options become complex and a bit bewildering, because the problem itself is complicated. How do you enable developers to deploy new versions of shared components, while at the same time ensuring that the new version won't break an application that depends on behavior provided by a prior version? The only answer is to develop a system that allows differing versions of the same component to be deployed *side-by-side.*

The first thing you need to learn about deployment in the .NET Framework is that the days of deploying shared components to `C:\Windows\System32` are gone. Instead, the .NET Framework includes two kinds of assemblies, well-defined rules governing how the system searches for them, and an infrastructure that supports the deployment of multiple versions of the same assembly on the same machine.

Before talking about the two kinds of assemblies, I will explain the two kinds of deployment:

Public, or Global A publicly deployed assembly is one you intend to share either among your own applications or with applications written by other people. The .NET Framework specifies a well-known location in which you are to deploy such assemblies.

Private A privately deployed assembly is one you do not intend to share. It is typically deployed to your application's base installation directory.

The type of assembly you create depends on how you intend to deploy it. The two types of assemblies are as follows:

Strong Name Assemblies A strong name assembly is digitally signed. The signature consists of the assembly's identity (name, version, and optional culture plus a key pair with public and private components). All global assemblies must be strong name assemblies. The name, version, and culture are attributes of the assembly, stored in its metadata. How you create and update these attributes depends largely on your development tools.

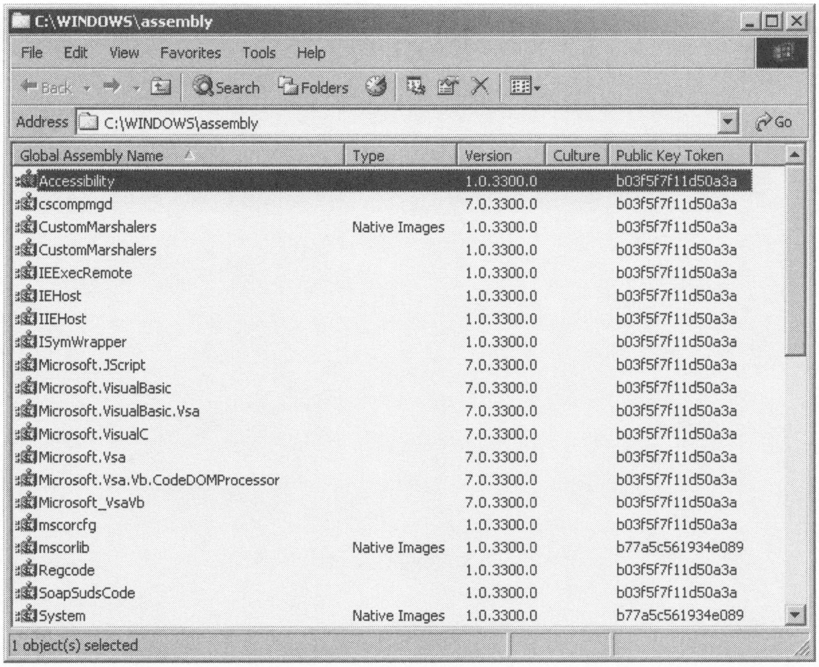
Everything Else There is no official name for an assembly that does not have a strong name. An assembly that does not have a strong name can only be deployed privately, and it cannot take advantage of the side-by-side versioning features of the CLR.

The key pair that completes the signing of a strong name assembly is generated by a tool provided by the .NET Framework SDK, called SN (`sn.exe`). SN creates a key file, which is referenced by the assembly in an attribute. The signing of an assembly can only take place when it is created, or *built.*

However, in the form of signing called *delayed signing*, developers work only with the public part of the key. Later, when the final build of the assembly is prepared, the private key is added to the signature. Currently the Delphi for .NET compiler is weak on custom attributes. Because attributes are required to express the identity and reference the key file, support for strong name assemblies isn't quite complete at the time of writing.

Strong name assemblies are usually intended to be shared, so you deploy them in the Global Assembly Cache (GAC) (but they can be deployed privately as well). Because the GAC is a system folder with a specialized structure, you can't just copy the assembly there. Instead, you must use another tool contained in the .NET Framework runtime, called GACUTIL (`gacutil.exe`). GACUTIL installs files into and removes files from the GAC.

The GAC is a special system folder with a hierarchy that supports side-by-side deployment of assemblies, as you can see in Figure 24.4. The intent is to hide the structure of the GAC so you don't have to worry about the complexities involved. GACUTIL reads the assembly's identity from its metadata and uses that information to construct a folder within the GAC to hold the assembly.


C:\WINDOWS\assembly
File Edit View Favorites Tools Help
Back Search Folders
Address C:\WINDOWS\assembly
Go

Global Assembly Name	Type	Version	Culture	Public Key Token
Accessibility		1.0.3300.0		b03f5f7f11d50a3a
cscompmgd		7.0.3300.0		b03f5f7f11d50a3a
CustomMarshalers	Native Images	1.0.3300.0		b03f5f7f11d50a3a
CustomMarshalers		1.0.3300.0		b03f5f7f11d50a3a
IEExecRemote		1.0.3300.0		b03f5f7f11d50a3a
IEHost		1.0.3300.0		b03f5f7f11d50a3a
IIEHost		1.0.3300.0		b03f5f7f11d50a3a
ISymWrapper		1.0.3300.0		b03f5f7f11d50a3a
Microsoft.JScript		7.0.3300.0		b03f5f7f11d50a3a
Microsoft.VisualBasic		7.0.3300.0		b03f5f7f11d50a3a
Microsoft.VisualBasic.Vsa		7.0.3300.0		b03f5f7f11d50a3a
Microsoft.VisualC		7.0.3300.0		b03f5f7f11d50a3a
Microsoft.Vsa		7.0.3300.0		b03f5f7f11d50a3a
Microsoft.Vsa.Vb.CodeDOMProcessor		7.0.3300.0		b03f5f7f11d50a3a
Microsoft_VsaVb		7.0.3300.0		b03f5f7f11d50a3a
mscorcfg		1.0.3300.0		b03f5f7f11d50a3a
mscorlib	Native Images	1.0.3300.0		b77a5c561934e089
Regcode		1.0.3300.0		b03f5f7f11d50a3a
SoapSudsCode		1.0.3300.0		b03f5f7f11d50a3a
System	Native Images	1.0.3300.0		b77a5c561934e089

1 object(s) selected

FIGURE 24.4
.NET's Global Assembly Cache as see in Windows Explorer

NOTE *You can see the true folder hierarchy by opening a command window and navigating to the* `C:\Windows\assembly` *directory. This hierarchy is hidden from you if you use Windows Explorer to view it.*

When you create a reference to a strong name assembly in the GAC, you are creating a reference to a specific version of that assembly. If a later version of that same assembly is deployed on the machine, it will be put into a separate location in the GAC, leaving the original intact. Because your assembly referenced a specific version, that's the version it always gets.

What's Next?

Your reaction to the .NET initiative will depend on your programming background. If you come from the Java environment, from a traditional Windows tool (even if object-oriented), or from Delphi, you may find many new features or many concepts you are already familiar with. The .NET platform is a technical achievement that offers a staggering array of options and possibilities. Compared with the earlier Microsoft system interfacing technologies (the Windows API and COM), the progress is significant. The best part is that you can take advantage of the platform using the tools and languages you feel are best for the job, soon including Delphi.

In Chapter 25 we will examine specific changes to the Delphi language. We will look at deprecated features and types, new features that have already been added to the language, and examples demonstrating the use of .NET Framework classes from Delphi for .NET.

Chapter 25

Delphi for .NET Preview: The Language and the RTL

THE LAST CHAPTER INTRODUCED Microsoft's .NET architecture. Now it is time to focus on the Delphi for .NET Preview that ships with Delphi 7. This chapter will cover specific changes that were made to the Delphi language to make it compatible with the Common Language Runtime. There have been some important additions to the language, such as namespaces and new visibility specifiers. Other long-standing features of the language had to be dropped because they are not supported in a type-safe environment.

This chapter discusses areas of the new compiler including the use of some of Microsoft's own class libraries for .NET and ASP support.

Bear in mind that the Delphi for .NET compiler is still being crafted as I write this. Borland's plan was to release the Preview compiler with Delphi 7 and to provide periodic updates to registered Delphi customers throughout the remainder of 2002 and in the early 2003 (before the Galileo project is completed). Some of the language features discussed here may not be implemented, or may be in varying states of completeness, depending on which version of the compiler you are working with. At the end of this chapter I list web resources you can monitor to keep abreast of the changes and latest news regarding Borland's Delphi for .NET compiler.

NOTE *As was true in Chapter 24, most of this material comes from John Bushakra.*

This chapter covers the following topics:

- Changes to the Delphi language
- Deprecated and new language features
- The run-time library and the VCL
- Using Microsoft's libraries
- ASP.NET with the Delphi language

Deprecated Delphi Language Features

We will begin by looking at some of the Delphi features that had to be dropped (deprecated) in order to be compatible with the Common Language Runtime (CLR). Then we'll look at new features added (or planned) in Delphi for .NET, which will shape the Delphi language in the near future, possibly also on the Win32 and Linux platforms.

Deprecated Types

Some of Delphi's types will not survive the transition to a managed, virtual execution system. So far, the following types are either known to be deprecated or have uncertain fates:

Pointers Pointers are considered unsafe types by the CLR, and all forms of pointer arithmetic are forbidden. *Unsafe* means the code cannot be verified for type safety. The final version of the Delphi for .NET compiler may support unsafe, unmanaged pointers, but in the meantime you can use dynamic arrays to get back some of the functionality of the (also deprecated) `GetMem`, `FreeMem`, and `ReallocMem` functions.

Types Based on *file of <type>* File types based on the old Pascal `file of <type>` syntax cannot be supported because the compiler has no way to determine the size of a given type on the target platform.

Pre-Delphi *Object* Syntax The pre-Delphi `object` syntax has been deprecated and will not be supported in the final release of the compiler. This syntax was introduced back in the Turbo Pascal era and allowed you to declare a new class with the syntax `type MyClass = object;`. Variables of this type were stack-based, contrary to the heap-based class type objects.

Real48 and Comp These types won't be supported. Real48 type is a 6-byte floating-point type. The Comp type will be replaced by Int64 in the future, as noted in the Delphi 7 Language Reference.

Strings and Other Types

The following types, although not candidates for deprecation, will have changes made to their underlying implementation. These changes are almost transparent, but you can expect slightly different behavior in some circumstances:

Strings In Delphi for .NET, strings map to the CLR type `System.String` and are wide by default. This means they use 16 bits per character, like the WideString type in Delphi 7. In addition, all characters are wide by default.

Records Records are mapped to value types. Chapter 24, "The Microsoft .NET Architecture from the Delphi Perspective," talked about the two main categories of types specified by the Common Type System (CTS): reference types and value types. A record will be a value type on the .NET platform. The CLR demands that value types cannot have inheritance, but you can define methods on them (which, of course, is completely new to the Delphi language). Methods defined on value types must be declared as final. (The new `final` keyword is discussed in "New Delphi Language Features" section.)

TDateTime In Delphi, this type is based on the same implementation as Microsoft's DATE type (see Chapter 2, "The Delphi Language," for more details). The .NET platform uses a different implementation. The `System.DateTime` structure (a descendant of System.ValueType) measures time from midnight, January 1, 0001 C.E. (Common Era) to 11:59:59 P.M., December 31, 9999 C.E. On this clock, one tick equals 100 nanoseconds. Going forward, Delphi for .NET will transition to the .NET platform standard for measuring time. Date calculations that depend on the floating-point implementation will require your attention when porting to the .NET platform. In particular, if you are using Delphi's `Trunc` and `Frac` functions to separate the date and time portions of the floating-point value, then you could be in for some interesting bugs in the future.

Currency Currency will be mapped to the CLR type System.Decimal.

Deprecated Code Features

As is the case with the types listed in the previous section, some Delphi features that have been part of the language a long time cannot be ported to the .NET platform:

Variant Records Variant records with overlapping fields are not supported by the CLR. In general, you can't make any assumptions regarding the layout of fields in a record declaration, because the Just In-Time (JIT) compiler reserves the right to optimize things to suit the underlying platform.

ExitProc Things don't always happen when you'd like them to, or in the order you'd like them to. Such problems with unit initialization and finalization have been overcome (although there are still issues to be aware of, as I discuss later in this chapter), but `ExitProc` is not supported.

Dynamic Aggregation of Interfaces The CLR doesn't support dynamic aggregation of interfaces using the `implements` keyword, because it cannot be verified for type safety. A class must declare all interfaces it will implement.

***ASM* Statements** ASM statements and inline assembly language are not supported by the Delphi for .NET Preview compiler. The future of `asm` in the final version is doubtful. The compiler would have to be able to mix managed IL (Intermediate Language) and unmanaged native CPU instructions, like Microsoft's Visual C++ compiler.

***automated* Keyword** The `automated` keyword was created to support OLE Automation. It is not needed in the .NET environment. The same is true for the `dispid` keyword, which is used to dispatch COM Automation methods by number instead of by name. Notice that although they're no longer explicitly required, GUIDs are supported on the .NET platform; they appear as custom attributes on a type.

Direct Memory Access Functions Direct memory access functions like `BlockRead`, `BlockWrite`, `GetMem`, `FreeMem`, and `ReallocMem`, as well as `Absolute`, and `Addr`, all deal with unmanaged pointers, and so cannot be used with managed, safe code. The @ operator is available in the current Preview version of the compiler (but is not expected to remain in the final version), although you cannot type cast pointers or do any pointer arithmetic.

NOTE *As discussed in Chapter 2, Delphi 7 provides a new set of compiler warnings to help you get ready to port your code. These warnings flag certain features and language constructs that are known to be unsafe on the .NET platform and therefore should be avoided. The warnings are turned off by default for a new Delphi 7 project but are active when you recompile an existing project. You can also turn them on with the* `{$WARN UNSAFE_CODE ON}` *compiler directive and similar directives. Refer to Chapter 2 for more details.*

New Delphi Language Features

The first release of the dccil compiler included new features required by the CLR, and more have been added in subsequent updates.

Unit Namespaces

Namespaces play an important role in the .NET Framework. They allow the class hierarchy to be extended by multiple third parties without fear of conflicting symbol names. Windows and COM use a 16-byte GUID to uniquely identify components, and this magic number must be recorded in the system registry. On the .NET platform, the concept of namespaces—plus metadata and the hard-and-fast rules about locating assemblies—makes GUIDs obsolete.

Ironically, the idea of a Delphi unit is similar to the CLR's namespaces. It's not too far a leap, if you think of a unit as a container of symbols, and a namespace as a container of units. In Delphi for .NET, the namespace to which a unit belongs is declared in the `unit` clause:

```
unit NamespaceA.NamespaceB.UnitA;
```

The dots indicate the containment of one namespace within another, and ultimately of the unit within the namespace. The dots separate the declaration into components, and each component—up to but not including the rightmost one—is a namespace. The entire declaration taken as a whole, dots and all, is the unit name. The dots simply serve as separators; no new symbols are introduced by the declaration. In this example, `NamespaceA.NamespaceB` is the namespace, and `NamespaceA.NamespaceB.UnitA` is the name of the unit. `NamespaceA.NamespaceB.UnitA.pas` would be the name of the source file, and the compiler would produce an output file called `NamespaceA.NamespaceB.UnitA.dcuil`.

The `program` statement (and eventually the `package` and `library` statements) optionally declares the default namespace for the entire project. Otherwise, the project is called a *generic project,* and the default namespace is that specified by the `-ns` compiler option. If no default project namespace is specified with compiler options, then behavior reverts to not using namespaces, like in Delphi 7 (and prior releases).

The `unit` clause does not have to declare membership in any explicit namespace. It might look like a traditional Delphi statement:

```
unit UnitA;
```

A unit that does not declare membership in a namespace is called a *generic unit.* Generic units automatically become members of the project namespace. Note, however, that this does not affect the source filename.

WARNING *At the time of this writing, namespace support is very limited; this section describes how it should work in the future, rather than how it works now.*

In the project file, you can specify a `namespaces` clause to list a set of namespaces for the compiler to search when it is trying to resolve references to generic units. The `namespaces` clause must appear immediately after the `program` (or `package` or `library`) statement and before any other clause or block type. The namespaces are separated by commas, and the list is terminated with a semicolon. For example:

```
program NamespaceA.MyProgram
  namespaces Foo.Bar, Foo.Frob, Foo.Nitz;
```

This example adds the namespaces `Foo.Bar`, `Foo.Frob`, and `Foo.Nitz` to the generic unit search space.

This discussion leads up to showing you how the compiler searches for generic units when you build your program. When you use a unit and fully qualify its name with the full namespace declaration, there is no problem:

```
uses Foo.Frob.Gizmos;
```

The compiler knows the name of the `dcuil` file (or the `.pas` file) in this case. But suppose you only said the following:

```
uses Gizmos;
```

This is called a *generic unit reference,* and the compiler must have a way to find its `dcuil` file.

The compiler searches namespaces in the following order:

1. The current unit namespace (if any)
2. The default project namespace (if any)
3. The namespaces listed in the project's `namespaces` clause (if any)
4. The namespaces specified by compiler options

For the first item, if the current unit specifies a namespace, then subsequent generic unit references in the current unit's `uses` clause are looked for first in the current unit's namespace. Consider this example:

```
unit Foo.Frob.Gizmos;
uses doodads;
```

The first search location for the unit `doodads` would be in the namespace `Foo.Frob`. So, the compiler would try to open `Foo.Frob.Doodads.dcuil`. Failing this, the compiler would move on and prefix the unit name `doodads` with the default project namespace, and so on down the list.

The same symbol name can appear in different namespaces. When such ambiguity occurs, you must refer to the symbol by its full namespace and unit name. If you have a symbol named `Hoozitz` in unit `Foo.Frob.Gizmos`, you can refer to the symbol with either

```
Hoozitz; // if the name is unambiguous
Foo.Frob.Gizmos.Hoozitz;
```

but not with

```
Gizmos.Hoozitz;       // error!
Frob.Gizmos.Hoozitz; // error!
```

Unit and namespace names can become quite long and unwieldy. You can create an alias for the fully qualified name with the `as` keyword in the `uses` clause:

```
uses Foo.Frob.DepartmentOfRedundancyDepartment.UIToys as ToyUnit;
```

Unit aliases introduce new identifiers, so their names cannot conflict with any other identifiers in the same unit (aliases are local to their unit). Even if you declare an alias, you can still use the original, longer name to refer to the unit.

NOTE *The case of a namespace declaration is preserved and emitted into assembly metadata as is. However, as far as Delphi is concerned, two namespaces that differ only in case are equivalent.*

Extended Identifiers

The cross-language integration of the CTS and CLR brings up some interesting situations for compiler developers. For example, what if the name of an identifier in an assembly is the same as one of your language keywords? Consider the Delphi language keyword `type`. `Type` is also the name of a CLR class. Because `type` is a language keyword, it cannot be used as the name of an identifier. You can avoid this problem two ways in Delphi for .NET (these techniques were not implemented in Delphi 7 and previous versions).

First, you can use the fully qualified name of the identifier:

```
var
  T: System.Type;
```

The second, shorter way is to use the new ampersand operator (`&`) to prefix the identifier. The following has the same effect as the previous example:

```
var
  T: &Type;
```

In this statement the ampersand tells the compiler to look for a symbol with the name `Type` and to not consider `it` as a keyword. The compiler will look for the `Type` symbol in the available units, finding it in System (the same mechanism works regardless of the unit defining the symbol).

The *final* and *sealed* Keywords

Two more concepts specified by the Common Language Infrastructure (CLI) have been added to the Delphi for .NET compiler: the class attribute `sealed` and the method attribute `final`. Putting the `sealed` attribute on a class effectively ends the class's ability to be used as a base class. Here is a sample code snippet:

```
type
  TDeriv1 = class (TBase)
    procedure A; override;
  end sealed;
```

A class cannot derive from a class that has been sealed. Similarly, a virtual method marked with the `final` attribute cannot be overridden in any descendant class, as in the following sample code.

```
type
  TDeriv1 = class (TBase)
    procedure A; override; final;
  end;

  TDeriv2 = class (TDeriv1)
    procedure A; override; // error: "cannot override a final method"
  end;
```

Borland added the `sealed` and `final` keywords to map an existing feature of .NET, but why did Microsoft introduce these attributes? The `final` and `sealed` attributes give users of your code important insights into how you intend your classes to be used. Moreover, these attributes give the compiler hints that allow it to generate more efficient Common Intermediate Language (CIL).

New Visibility and Access Specifiers

Delphi's notion of visibility—public, protected, and private—is a bit different from that of the CLI. In languages like C++ and Java, when you specify a visibility of private or protected on a class member, that class member is only visible to descendants of the class in which it is defined. As you saw in Chapter 2, however, Delphi enforces the idea of private and protected only for classes in different units, because everything is visible within a single unit. To be CTS compliant, the language required new visibility specifiers:

class private A member declared with class private visibility follows the C++ and Java rules. That is, class private members can be accessed only in methods or properties of the declaring class. Procedures and functions declared at the unit level and methods of other classes do not have access.

class protected Similarly, class protected members are visible only within the declaring class, and to descendants of the declaring class. Other classes in the same unit have access only if they inherit from this class.

See the ProtectedPrivate example in the `LanguageTest` folder of the chapter's source code for a trivial test case.

Class Static Members

Delphi has long supported class methods—methods you can apply to a class as a whole and also as a specific instance, even if the methods' code cannot refer to the current object (the `Self` parameter of a class methods references the current class, not the current object). Delphi for .NET extends this idea by adding the `class static` specifier, class properties, class static fields, and class constructors:

Class Static Methods Like Delphi 7 class methods, class static members can be called without an object instance, and no `Self` parameter refers to an object. Unlike in Delphi 7, however, you cannot refer to the class itself. For example, calling the `ClassName` method will fail. Also unlike in Delphi 7, you cannot use the `virtual` keyword with class static methods.

Class Static Properties Like class methods, class static properties can be accessed without an object instance. The access methods or backing fields for class static properties must be declared class static themselves. Class static properties cannot be published, nor can they have stored or default value definitions.

Class Static Fields A class static field can be accessed without an object instance. Class static fields and properties are typically used as design tools; they allow you to declare variables and constants within the meaningful context of a class declaration.

Class Constructor A class constructor is a private constructor (it must be declared with class private visibility) that runs prior to the first use of the declaring class. The CLR offers no guarantee of when this will happen, except to say it will happen before the first use of the class. In CLR terms, this can get a bit tricky, because code is not considered "used" unless (and until) it is executed. A class can declare only one class constructor. Descendants can declare their own class constructors, but only one can be declared in any class.

You can't call a class constructor from source code; it is called automatically as a way to initialize class static fields and properties. Even the `inherited` keyword is prohibited, because the compiler takes care of this for you.

The following example class declaration illustrates the syntax for these new specifiers:

```
TMyClass = class
class private // can only be accessed within TMyClass
  // Class constructor must have class private visibility
  class constructor Create;
class protected // can be accessed in TMyClass and in descendants
  // Class static accessors for class static property P1, below
  class static function getP1 : Integer;
  class static procedure setP1(val : Integer);
public
  // fx can be called without an object instance
  class static function fx(p : Integer) : Integer;
  // Class static property P1 must have class static accessors
  class static property P1 : Integer read getP1 write setP1;
end;
```

Nested Types

Nested types are similar to class fields, in that they can be accessed through a class reference; an object instance is not needed. Declared within the scope of a class, nested types give you a way to use the enclosing class as a kind of namespace for the type.

Multicast Events

Delphi has always had the ability to set an event listener—a function that is called when an event is fired. The CLR supports the use of multiple event listeners so that more than one function can respond when an event is fired. These are called *multicast events*. Delphi for .NET introduces two new property access methods, `add` and `remove`, to support multicast events. The `add` and `remove` methods can be used only on properties that are events.

To support multicast events, you must have a way to store all the functions that register themselves as listeners. As stated in Chapter 24, multicast events are implemented using the CLR `MulticastDelegate` class. And, as discussed there, the compiler hides a lot of complexity behind the scenes. The `add` and `remove` keywords handle the storage and removal of event listeners, but the containment mechanism is an implementation detail you aren't expected to deal with. The compiler automatically generates `add` and `remove` methods for you, and these methods implement storage of event listeners in an efficient way.

In the final release of Delphi for .NET, the `add` and `remove` methods should work hand in hand with an overloaded version of the standard functions `Include` and `Exclude`. In your source code, when you'd want to register a method as an event listener, you call `Include`. To remove a method, call `Exclude`. For example:

```
Include(EventProp, eventHandler);
Exclude(EventProp, eventHandler);
```

Behind the scenes, `Include` and `Exclude` will call the methods assigned to the `add` and `remove` access functions, respectively. At the time of this writing, this technology wasn't working, so the book examples don't use it.

To support legacy code, the Delphi assignment operator (`:=`) still works as a way to assign a single event handler. The compiler generates code to go back and replace the last event handler (and only that event handler) that was set with the assignment operator. The assignment operator works separately and independently from the `add`/`remove` (or `Include`/`Exclude`) mechanism. In other words, the use of the assignment operator does not affect the list of event handlers that have been added to the `MulticastDelegate`.

As an example, you can refer to the XmlDemo program. The following code snippet (the working code at the time of this writing) creates a button at run time and installs two event handlers for its `Click` event:

```
MyButton := Button.Create;
MyButton.Location := Point.Create (
  Width div 2 - MyButton.Width div 2, 2);
MyButton.Text := 'Load';
MyButton.add_Click (OnButtonClick);
MyButton.add_click (OnButtonClick2);
Controls.Add (MyButton);
```

Custom Attributes

Recall from Chapter 24 that one of the requirements of the CLI is an extensible metadata system. All .NET language compilers are required to emit metadata for the types defined within an assembly. The *extensible* part of extensible metadata means that programmers can define their own attributes and apply them to just about anything: assemblies, classes, methods, and more. The compiler emits these into the assembly's metadata. At run time, you can query for the attributes that were applied to an entity (assembly, class, method, and so on) using the methods of the CLR class `System.Type`.

Custom attributes are reference types derived from the CLR class `System.Attribute`. Declaring a custom attribute class is just like declaring any other class (this code snippet is extracted from the trivial NetAttributes project part of the LanguageTest folder):

```
type
  TMyCustomAttribute = class(TCustomAttribute)
  private
    FAttr : Integer;
  public
    constructor Create(val: Integer);
    property customAttribute : Integer read FAttr write FAttr;
  end;
  ...
  constructor TMyCustomAttribute.Create(val: Integer)
  begin
    inherited Create;
    customAttribute := val;
  end;
```

The syntax for applying the custom attribute is similar to that of C#:

```
type
  [TMyCustomAttribute(17)]
  TFoo = class
  public
    function P1(X : Integer) : Integer;
  end;
```

The custom attribute is applied to the construct immediately following it. In the example, it is applied to the class `TFoo`. No doubt you noticed that the custom attribute syntax is nearly identical to that of Delphi's GUID syntax. Here we have a problem: GUIDs are applied to interfaces; they must immediately *follow* the interface declaration. Custom attributes, on the other hand, must immediately *precede* the declaration to which they apply. How can the compiler determine whether the thing in the square brackets is a traditional Delphi-style GUID (which should be applied to the preceding interface declaration) or a .NET-style custom attribute (which should be applied to the first member of the interface)?

There is no way to tell, so you have to punt—make a special case for custom attributes and interfaces. If you apply a GUID to an interface, it must immediately follow the declaration of the interface, and it must follow the established Delphi syntax:

```
type
  interface IMyInterface
  ['(12345678-1234-1234-1234-1234567890ab)']
```

CLR's `GuidAttribute` custom attribute is used to apply GUIDs; it is part of the `System.Runtime.InteropServices` namespace. If you use this custom attribute to apply a GUID, then you must follow the CLR standard and put the attribute declaration before the interface.

Class Helpers

Class helpers are an intriguing new language feature added to Delphi for .NET. The main reason for supporting class helpers is the way Borland maps .NET core classes with its own RTL classes, as covered later in the section "Class Helpers for the RTL." Here I will focus on this feature from a language perspective.

A class helper gives you a way to extend a class without using derivation, by adding new methods (but not new data). The odd fact, compared to inheritance, is that you can create objects of the original class, which is extended maintaining the same name. This means you can *plug-in* methods to an existing object of an existing class. A simple example will help clarify the idea.

Suppose you have a class (probably one you haven't written yourself—otherwise you could have extended it right away) like this:

```
type
  TMyObject = class
  private
    Value: Integer;
    Text: string;
  public
    procedure Increase;
  end;
```

Now you can add a `Show` method to objects of this class by writing a class helper to extend it:

```
type
  TMyObjectHelper = class helper for TMyObject
  public
    procedure Show;
  end;

procedure TMyObjectHelper.Show;
begin
  WriteLn (Text + ' ' + IntToStr (Value) + ' -- ' +
    Self.ClassType.ClassName + ' -- ' + ToString);
end;
```

Notice that `Self` in the class helper method is the object of the class that the helper is for. You can use it like this:

```
Obj := TMyObject.Create;
...
Obj.Show;
```

You'll end up seeing the name of the `TMyObject` class in the output. If you inherit from the class, however, the class helper will also be usable on the derived class (so you end up adding a method to an entire hierarchy), and everything will work properly. For your experiments, refer to the ClassHelperDemo example in the `LanguageTest` folder.

The Run-time Library and VCL

Located in the `source\rtl` directory of the Delphi for .NET Preview installation, you will find the source files for the run-time library (RTL). You can already see the migration of units into CLR namespaces, as reflected in the source filenames.

Borland is taking the approach (for the most part) of preserving the original unit name and prefixing it with the namespace name `Borland.Delphi`. Operating system–specific things (such as registry and `ini` file utilities) go in the `Borland.Win32` namespace, because these classes, procedures, and functions are Borland-specific wrappers of Windows-specific features. The naming trend should continue, although not all units will make the transition, and some will have their contents reorganized into an appropriate namespace.

Perusing the RTL source files is both highly educational and highly recommended; however, remember that you are looking at a preview release of a product, not the final version. The contents of the RTL source files are still subject to change—you should not make any assumptions, and you should definitely not introduce dependencies into your own code based on what you see there.

Class Helpers for the RTL

With the warnings out of the way, let's see what has been done to the RTL so far. The most interesting change is perhaps the introduction of class helpers.

`Borland.Delphi.System.pas` includes the following declaration:

```
type
  TObject = System.Object;
```

It tells you that Delphi's `TObject` class is an alias for the CLR class `System.Object`. This is important: `TObject` is not a descendant of `System.Object`—it is semantically equivalent. What happened to the methods that used to be defined in `TObject`, such as `ClassName` and `ClassParent`? That's where the class helper comes in.

The methods that used to be directly declared and implemented in `TObject` are now declared and implemented in a class called `TObjectHelper`. `TObjectHelper` is then declared to be a class helper for `TObject`. In `Borland.Delphi.System.pas` is the following:

```
type
  TObjectHelper = class helper for TObject
    procedure Free;
    function ClassType: TClass;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: TObject;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: string): TObject;
    class function SystemType: System.Type;
    function FieldAddress(const Name: string): TObject;
    procedure Dispatch(var Message);
  end;
```

A class helper gives you a way to extend a class without using derivation.

You might want to extend a CLR class but not derive from it in order to use the CLR class with existing Delphi code. No doubt you've noticed that Delphi's class framework and the .NET Framework share a fair amount of functionality. In some cases there are name clashes between the two—for example, Borland's `Exception` class and the CLR's `System.Exception` class. On one hand, the two classes do basically the same thing, but they expose that functionality in different ways. On the other hand, large amounts of existing Delphi code have been using Borland's `Exception` class for a long time.

The only workable solution was to create a mechanism that would allow developers (including Borland) to leverage the CLR classes, and that would also allow the CLR classes to be extended to include long-standing behaviors expected by existing Delphi code.

The VCL

The .NET Framework classes in the `System.Windows.Forms` namespace are not a replacement for the GUI portion of the Win32 API. The same happens with most other portions of the .NET Framework: its role is to make the underlying API easier to use by providing an easier object-oriented interface compatible with the core services of the .NET environment. The GUI subset of Win32 is still there, occupying the same place it always has. `System.Windows.Forms` organizes this GUI subset of the Win32 API, presents it in an object-oriented way, and layers an event model on top of it; but classes in `System.Windows.Forms` call the unmanaged code in Win32. When you use `System.Windows.Forms`, you are still calling the Win32 APIs, but now you have a large layer of software called the CLR sitting between your code and Win32.

This entire preamble is important to acknowledge why the VCL takes this same approach. `TObject` is rooted (if you will) from `System.Object` through the use of a class helper. `TPersistent` and `TComponent` still descend from there. Thus the VCL class `TForm`, for example, is not a descendant of the class `System.Windows.Forms.Form`. Instead, the entire VCL hierarchy remains much as it is today. `TForm` will ultimately descend from `TWinControl`, which itself is a descendant of `TControl` and then `TComponent`.

If you consider the entire `System.Windows.Forms` namespace as a single entity, the VCL then becomes a kind of sibling to it, rather than a child of it. Both frameworks ultimately rely on the native, unmanaged Win32 API for the underlying implementation of the user interface controls.

Looking into the VCL.NET Source

The update of the Delphi .NET Preview compiler made available by Borland in November 2002, although still preliminary, provides details about the architecture the company is planning. If you open the Borland.Vcl.Controls unit, you'll be surprised by its similarity to the Win32 version. The source code is almost identical; the differences exist behind the scenes at the `TObject` and `TComponent` level. I've already covered the former, so let's focus on the core component class, which is defined in three steps:

```
type
  TComponent = System.ComponentModel.Component;
  TComponentHelper = class helper (TPersistentHelper) for TComponent
  TComponentSite = class(TObject, ISite, IServiceProvider)
```

The `TComponent` class corresponds to the .NET `Framework` class, with a helper providing extra methods and properties and a further class offering the extra data required by the helper class. The situation is complex, and I don't want to get into the details because the `TComponent` class is marked as experimental and may change in further updates.

Getting back to the VCL, a large set of components is already available, so you can begin porting code. The only trouble you'll face with the November 2002 update is that streaming is not supported; so, you must add the component-creation code in the form constructor (an action that will not be required by the final version of Delphi for .NET).

As an interesting example to help you figure out the architecture of the VCL classes, I've ported to .NET the ClassInfo example from Chapter 3, "The Run-Time Library." The NetClassInfo example uses this modified code in the project source (again, something you won't have to do in the future):

```
Application.Initialize;
Form1 := TForm1.Create (Application);
Application.MainForm := Form1;
```

The code for the form, as I've mentioned, has an extra method called by the constructor and used to initialize the controls. This method is quite long, so I'll provide only a few excerpts here:

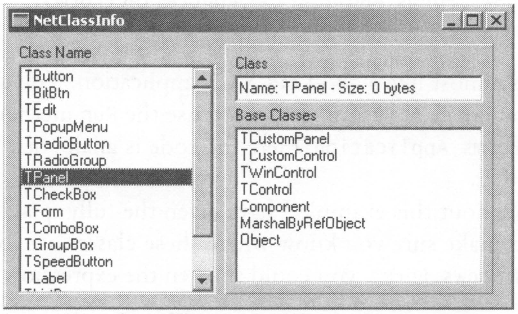
```
procedure TForm1.InitializeControls;
begin
  // creating all controls...
  Label3:= TLabel.Create(Self);
  Panel1:= TPanel.Create(Self);
  Label1:= TLabel.Create(Self);
  Label2:= TLabel.Create(Self);
  ...

  // setting form properties and events
  Left:= 217;
  Top:= 109;
  Caption:= 'Class Info';
  OnCreate:= FormCreate;

  // initializing controls (only one is listed here)
  with Label3 do
  begin
    Parent:= Self;
    Left:= 8;
    Top:= 8;
    Width:= 56;
    Height:= 13;
    Caption:= 'Class Name';
  end;
```

The rest of the application's code remains almost identical, which is surprising considering that this is a low-level example. I had to remove the call to `InstanceSize`, because the compiler cannot resolve the size of an object given the architecture of .NET, and I had to test for the base class against `Object` instead of `TObject`. Here is the code snippet that produces the output shown in Figure 25.1:

```
procedure TForm1.ListClassesClick(Sender: TObject);
var
  MyClass: TClass;
begin
  MyClass := ClassArray [ListClasses.ItemIndex];
  EditInfo.Text := Format ('Name: %s - Size: %d bytes',
    [MyClass.ClassName, 0 {MyClass.InstanceSize}]);
  with ListParent.Items do
  begin
    Clear;
    while MyClass.ClassName <> 'Object' do
    begin
      MyClass := MyClass.ClassParent;
      Add (MyClass.ClassName);
    end;
  end;
end;
```

FIGURE 25.1

The NetClassInfo example shows the base classes of a given component.



TButton
TBitBtn
TEdit
TPopupMenu
TRadioButton
TRadioGroup
TPanel
TCheckBox
TForm
TComboBox
TGroupBox
TSpeedButton
TLabel

Name: TPanel - Size: 0 bytes

TCustomPanel
TCustomControl
TWinControl
TControl
Component
MarshalByRefObject
Object

Further VCL Examples

To provide starting points for your own experiments with the VCL under .NET, I've built two more examples. NetEuroConv is a port of the EuroConv example from Chapter 3 based on the RTL's conversion engine. NetLibSpeed is a port of the LibSpeed example used in Chapter 5 ("Visual Controls") to compare the VCL and VisualCLX libraries' speed in creating visual components. The number you'll see makes little sense in such a preliminary version of a library, although the fact that VCL.NET takes four to five times as long for the same purpose may worry you.

As I mentioned, these examples are meant to be only starting points for your experiments. They may not work with further updates of the Delphi for .NET Preview.

NOTE *Stay tuned to my website for updates of this section of the book and the related examples.*

Using Microsoft Libraries

The VCL is not quite ready, but you can use the .NET Framework class library as a basis for experimentation with the Delphi for .NET Preview compiler. It can be educational to build programs with the compiler and then inspect them with Intermediate Language Disassembler (ILDASM), for instance. This will be the aim of this section. If you want to look at a simpler example using XML support, refer to the XmlDemo mentioned earlier in the chapter.

The CLRReflection program opens an assembly and then uses reflection to inspect the modules and types defined within that assembly. This program demonstrates using a common dialog box (the `OpenFileDialog`), constructing menus, handling events, using Delphi's dynamic arrays, and, of course, reflection. Let's look at the project file first:

```
program CLRReflection;

uses
  System.Windows.Forms,
  ReflectionUnit;

var
  reflectForm : ReflectionForm;
begin
  reflectForm := ReflectionForm.Create;
  System.Windows.Forms.Application.Run(reflectForm);
end.
```

The code looks almost like a good old VCL application. You define a variable for your main form, and then you create the form. Then you use the `Run` method of the .NET Framework class `System.Windows.Forms.Application`. Here the code is analogous (at least in concept) to the way it is done in the VCL.

Note that throughout this example I have given the fully qualified name for .NET Framework classes. I did so to make sure you know where these classes are located. Because the **uses** clause includes `System.Windows.Forms`, you could shorten the expression

```
System.Windows.Forms.Application.Run(reflectForm);
```

to

```
Application.Run(reflectForm);
```

Now, look at Listing 25.1, which shows the unit where the main form is defined. Note that this code compiles with the November 2002 update of the Delphi for .NET Preview, but not with the version originally shipping with Delphi 7.

LISTING 25.1: THE REFLECTIONUNIT UNIT OF THE CLRREFLECTION EXAMPLE

```
unit ReflectionUnit;

interface
```

```
uses
    System.Windows.Forms,
    System.Reflection,
    System.Drawing,
    Borland.Delphi.SysUtils;

type
  ReflectionForm = class(System.Windows.Forms.Form)
  private
    mainMenu: System.Windows.Forms.MainMenu;
    fileMenu: System.Windows.Forms.MenuItem;
    separatorItem: System.Windows.Forms.MenuItem;
    openItem: System.Windows.Forms.MenuItem;
    exitItem: System.Windows.Forms.MenuItem;

    showFileLabel: System.Windows.Forms.Label;
    typesListBox: System.Windows.Forms.ListBox;
    openFileDialog: System.WIndows.Forms.OpenFileDialog;
  protected
    procedure InitializeMenu;
    procedure InitializeControls;
    procedure PopulateTypes(fileName: String);
    { Event Handlers }
    procedure exitItemClick(sender: TObject; Args: System.EventArgs);
    procedure openItemClick(sender: TObject; Args: System.EventArgs);
  public
    constructor Create;
end;

implementation

constructor ReflectionForm.Create;
begin
  inherited Create;

  SuspendLayout;
  InitializeMenu;
  InitializeControls;

  { Initialize the form and other member variables }
  openFileDialog := System.Windows.Forms.OpenFileDialog.Create;
  openFileDialog.Filter := 'Assemblies (*.dll;*.exe)|*.dll;*.exe';
  openFileDialog.Title := 'Open an assembly';

  AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
  ClientSize := System.Drawing.Size.Create(631, 357);
  Menu := mainMenu;
  Name := 'reflectionForm';
  Text := 'Reflection in Delphi for .NET';
```

```
    { Add the controls to the form's collection. }
    Controls.Add(showFileLabel);
    Controls.Add(typesListBox);
    ResumeLayout;
  end;

  { Build the main menu }
  procedure ReflectionForm.InitializeMenu;
  var
    menuItemArray : array of System.Windows.Forms.MenuItem;
  begin
    mainMenu := System.Windows.Forms.MainMenu.Create;
    fileMenu := System.Windows.Forms.MenuItem.Create;
    openItem := System.Windows.Forms.MenuItem.Create;
    separatorItem := System.Windows.Forms.MenuItem.Create;
    exitItem := System.Windows.Forms.MenuItem.Create;

    { Initialize mainMenu }
    mainMenu.MenuItems.Add(fileMenu);

    { Initialize fileMenu }
    fileMenu.Index := 0;
    SetLength(menuItemArray, 3);
    menuItemArray[0] := openItem;
    menuItemArray[1] := separatorItem;
    menuItemArray[2] := exitItem;
    fileMenu.MenuItems.AddRange(menuItemArray);
    fileMenu.Text := '&File';

    // openItem
    openItem.Index := 0;
    openItem.Text := '&Open...';
    openItem.add_Click(openItemClick);

    // separatorItem
    separatorItem.Index := 1;
    separatorItem.Text := '-';

    // exitItem
    exitItem.Index := 2;
    exitItem.Text := 'E&xit';
    exitItem.add_Click(exitItemClick);
  end;

  { Create the controls and populate the form }
  procedure ReflectionForm.InitializeControls;
  begin
    { Initialize showFileLabel }
  showFileLabel := System.Windows.Forms.Label.Create;
  showFileLabel.Location := System.Drawing.Point.Create(5, 6);
  showFileLabel.Name := 'showFileLabel';
  showFileLabel.Size := System.Drawing.Size.Create(616, 37);
  showFileLabel.TabIndex := 0;
  showFileLabel.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right
  showFileLabel.Text := 'Showing types in: ';

  { Initialize typesListBox }
  typesListBox := System.Windows.Forms.ListBox.Create;
  typesListBox.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Bottom or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right;
  typesListBox.Location := System.Drawing.Point.Create(8, 46);
  typesListBox.Name := 'typesListBox';
  typesListBox.Size := System.Drawing.Size.Create(610, 303);
  typesListBox.Font := System.Drawing.Font.Create('Lucida Console', 8.25,
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, 0);
  typesListBox.TabIndex  := 1;
end;

{ Event handler for the Exit menu item }
procedure ReflectionForm.exitItemClick(sender: TObject; Args: System.EventArgs);
begin
  System.Windows.Forms.Application.Exit;
end;

{ Event handler for the Open menu item }
procedure ReflectionForm.openItemClick(sender: TObject; Args: System.EventArgs);
begin
  if openFileDialog.ShowDialog = DialogResult.OK then
  begin
    showFileLabel.Text := 'Showing types in: ' + openFileDialog.FileName;
    PopulateTypes(openFileDialog.FileName);
  end;
end;

{ Open the given assembly, and reflect over its modules }
{ and types.                                            }
procedure ReflectionForm.PopulateTypes(fileName : String);
var
  assy: System.Reflection.Assembly;
  modules: array of System.Reflection.Module;
  module: System.Reflection.Module;
  types: array of System.Type;
```

```
    t: System.Type;
    members: array of System.Reflection.MemberInfo;
    m: System.Reflection.MemberInfo;
    i,j,k: Integer;
    s: String;
  begin
    try
      { Clear the listbox }
      typesListBox.BeginUpdate;
      typesListBox.Items.Clear;

      { Load the assembly and get its modules }
      assy    := System.Reflection.Assembly.LoadFrom(fileName);
      modules := assy.GetModules;

      {For every module, get all types }
      for i := 0 to High(modules) do
      begin
        module := modules[i];
        types  := module.GetTypes;

        { For every type, get all of its members }
        for j := 0 to High(types) do
        begin
          t := types[j];
          members := t.GetMembers;

          { for every member, get type information and add to list box }
          for k := 0 to High(members) do
          begin
            m := members[k];
            s := module.Name + ':' + t.Name + ': ' + m.Name +
              ' (' + m.MemberType.ToString + ')';
            typesListBox.Items.Add(s);
          end;
        end;
      end;
      typesListBox.EndUpdate;
    except
      System.Windows.Forms.MessageBox.Show('Could not load the assembly.');
    end;
  end;

  end.
```

The unit begins by declaring its dependency on .NET Framework `dcuil` files and on the `Borland.Delphi.SysUtils` unit. From there it goes straight into declaring the class for the main form, which is a descendent of the .NET Framework class, `System.Windows.Forms.Form`. The form class layout looks familiar: You have member variables for all the controls, and these are declared to be of types found in the .NET Framework class library.

The functions `exitItemClick` and `openItemClick` are event handler declarations. The signature of event handler methods is specified by the CLR. All event handlers are procedures that take two parameters: the object that fired the event (a derivative of `System.Object`) and the event arguments, which are wrapped in the `System.EventArgs` (or a derived) class. (You will see how to hook up these event handlers in a moment.)

Let's move on to the class constructor. I must call attention to the first statement in the constructor, which calls `inherited Create`.

WARNING *Here you see a major departure from the .NET Framework way of life, compared to what you are used to with the VCL and with Delphi in general. In Delphi, the constructor initializes member variables, putting the object instance into a known-good state; it does not do any memory allocation. So, it is not uncommon to see a constructor make assignments and then call the inherited constructor. Indeed, you might not call the inherited constructor at all. In Delphi for .NET, you can't get away with this approach. In your constructor, you* must *call* `inherited Create`, *and it must be the method's first executable statement. Currently, if you fail to do so, you will get a compiler error saying that* `Self` *is uninitialized and that the inherited constructor must be called prior to accessing any ancestor fields.*

After calling the inherited constructor, you are back in familiar territory. Although this code uses a different class hierarchy, it should be clear to any Delphi programmer. You make an instance of `System.Windows.Forms.OpenFileDialog` by calling the `Create` constructor—this is how you create an instance of any .NET Framework class.

The next few lines demonstrate setting properties, both of the `OpenFileDialog` object instance and of the form itself. Finally, you add two controls (a label for the filename and the ListBox that will hold the assembly) to the form's `Controls` collection, which is a property of type `Control.ControlCollection`.

The `InitializeMenu` procedure demonstrates allocation and layout of a `System.Windows.Forms.MainMenu` object instance. Where the File menu is initialized, a dynamic array holds each menu item. The dynamic array is then passed to the `AddRange` method. This code could have been accomplished by calling the `Add` method separately for each menu item.

The next interesting thing in `InitializeMenu` is the wiring of the menu item event handlers. In Chapter 24 and earlier in this chapter, I mentioned the behind-the-scenes complexity involved with delegates and multicast events. Here you see some of that complexity coming to the foreground.

You can't do it yet in Delphi for .NET, but in other .NET languages such as C#, you can use the language keyword `event` to introduce an event handler delegate. The event declaration specifies a delegate to use as a callback mechanism. Because the event is a `System.MulticastDelegate` derivative (a `System.EventHandler` delegate in this case), other objects can add and remove event handlers, and these handlers are called when the event fires.

The C# language adds a bit of syntactic sugar to help this pill go down more easily. C# defines `+=` and `-=` operators for adding and removing event handlers, respectively. Eventually Delphi will get its own spoonful of sugar, with the `Include/Exclude` mechanism mentioned previously. CTS mandates that all .NET compilers targeting this event model must generate methods named `add_<Event>` and `remove_<Event>`. These `add_` and `remove_` methods wrap the `Combine` and `Remove` methods declared in `System.Delegate`.

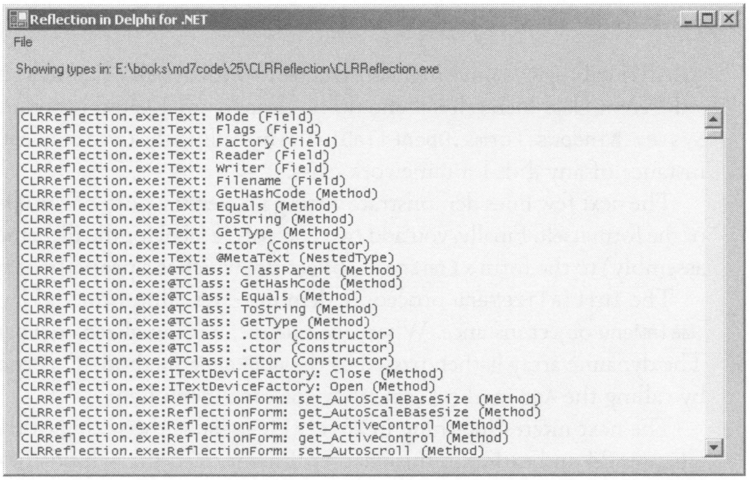
For now, to assign an event handler, you must use these `add_` and `remove_` methods; ordinarily, you would not concern yourself with them, because the compiler would hide this complexity. In the current class declaration, you introduce two methods whose signatures match the `System.EventHandler` delegate: `openItemClick` and `exitItemClick`. You then call the `add_Click` method on the respective menu item, passing your event handler as the callback method.

Now that the setup is out of the way, let's look at the code that reflects over the types defined within an assembly. You can load any assembly (thus creating an object instance), given its filename, with the static `LoadFrom` method. Once you have an assembly object, the keys to the kingdom are yours; you can use reflection to look over the assembly from any angle.

The collection of modules contained within an assembly is available with the `GetModules` method. From there you can drill down to the types defined in the module with `GetTypes`. As you saw in the `InitializeMenu` procedure, you can use dynamic arrays for properties that expose a collection with a `System.Array`.

Finally, each individual member of the module and types arrays contains a `Name` property, which you can use to build a string to display in the ListBox. The final effect of the code is visible in Figure 25.2.

FIGURE 25.2
The CLRReflection example, with an assembly loaded



Showing types in: E:\books\md7code\25\CLRReflection\CLRReflection.exe

```
CLRReflection.exe:Text: Mode (Field)
CLRReflection.exe:Text: Flags (Field)
CLRReflection.exe:Text: Factory (Field)
CLRReflection.exe:Text: Reader (Field)
CLRReflection.exe:Text: Writer (Field)
CLRReflection.exe:Text: Filename (Field)
CLRReflection.exe:Text: GetHashCode (Method)
CLRReflection.exe:Text: Equals (Method)
CLRReflection.exe:Text: ToString (Method)
CLRReflection.exe:Text: GetType (Method)
CLRReflection.exe:Text: .ctor (Constructor)
CLRReflection.exe:Text: @MetaText (NestedType)
CLRReflection.exe:@TClass: ClassParent (Method)
CLRReflection.exe:@TClass: GetHashCode (Method)
CLRReflection.exe:@TClass: Equals (Method)
CLRReflection.exe:@TClass: ToString (Method)
CLRReflection.exe:@TClass: GetType (Method)
CLRReflection.exe:@TClass: .ctor (Constructor)
CLRReflection.exe:@TClass: .ctor (Constructor)
CLRReflection.exe:@TClass: .ctor (Constructor)
CLRReflection.exe:ITextDeviceFactory: Close (Method)
CLRReflection.exe:ITextDeviceFactory: Open (Method)
CLRReflection.exe:ReflectionForm: set_AutoScaleBaseSize (Method)
CLRReflection.exe:ReflectionForm: get_AutoScaleBaseSize (Method)
CLRReflection.exe:ReflectionForm: set_ActiveControl (Method)
CLRReflection.exe:ReflectionForm: get_ActiveControl (Method)
CLRReflection.exe:ReflectionForm: set_AutoScroll (Method)
```

ASP.NET with the Delphi Language

You might like it or not (I'm not that fond of it), but Microsoft's ASP technology plays a significant role in the development of web applications, as least on the Windows platform. With the transition to ASP.NET, this technology has fully embraced the .NET Framework; now, with the availability of a Delphi compiler for .NET, the Delphi language can be your language of choice in the development of ASP applications.

To create a test, configure IIS to support ASP.NET (I won't cover these steps, which are beyond the scope of this book, but you can find more information on `www.asp.net`) and then place in the target folder the `web.config` file distributed by Borland along with the Delphi for .NET Preview (and available in the `aspx` subfolder). This configuration file defines the mapping of the language to a specific library, again provided by Borland. The core of the file (which uses XML format) has the following elements:

```
<compilation debug="true">
  <assemblies>
    <add assembly="DelphiProvider" />
  </assemblies>
  <compilers>
    <compiler language="Delphi" extension=".pas"
      type="Borland.Delphi.DelphiCodeProvider,DelphiProvider" />
  </compilers>
</compilation>
```

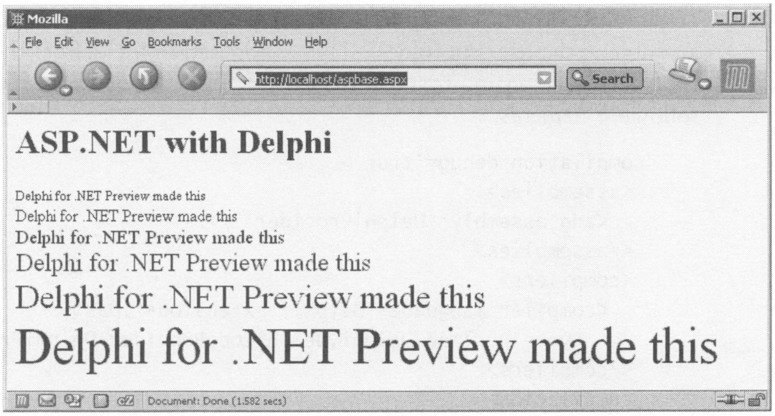
To test that the configuration is correct, nothing is better than trying an example. Create a new file (I've called mine `aspbase.aspx`, available in the `DelphiAsp` folder of the chapter source code) and type something like the following:

```
<html>
<body>

<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
procedure HelloMessage(msg: string);
var
  i: Integer;
begin
  for i := 2 to 7 do
    Response.Write ('<font size=' + inttostr (i) +
      '>' + msg + '</font> <br>')
end;
</script>

<% HelloMessage('Delphi for .NET Preview made this'); %>

</body>
</html>
```

The effect is to execute the Delphi code after transforming this file into a .NET source, compiling it with the Delphi preview compiler, and compiling the IL into assembly code (remember, even scripts in .NET are compiled before they are executed). If everything goes well, the browser should display output like that shown in Figure 25.3.

From this point on, you can do anything an ASP.NET application provides. The only other example I want to show you is the use of controls with event handlers, which I've covered in a different situation for .NET applications based on Windows forms.

FIGURE 25.3
The `aspbase.aspx` example in a browser


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost/aspbase.aspx
Search
Document: Done (1.582 secs)

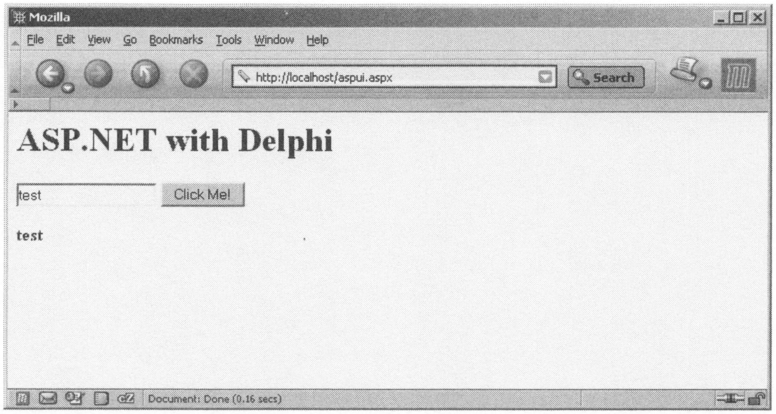
ASP.NET with Delphi

Delphi for .NET Preview made this
Delphi for .NET Preview made this
Delphi for .NET Preview made this
Delphi for .NET Preview made this
Delphi for .NET Preview made this
Delphi for .NET Preview made this

This example, saved in the `aspui.aspx` file in the `AspDelphi` folder, uses HTML to define a form with a textbox (that is, an edit control) and a button, plus an output label. The button has a Delphi language event handler attached to it, which moves the user input to the label (the output of the program appears in Figure 25.4):

```
<html>
<body>

<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
  procedure ButtonClick(Sender: System.Object; E: EventArgs);
  begin
    Message.Text := Edit1.Text;
  end;
</script>

<form runat="server">
   <asp:textbox id="Edit1" runat="server"/>
   <asp:button text="Click Me!" OnClick="ButtonClick" runat="server"/>
</form>

<p><b><asp:label id="Message" runat="server" text="message"/></b></p>

</body>
</html>
```

This has been a limited introduction to ASP.NET with the Delphi language provider. However, it should give you a feeling for the possibilities opening up for Delphi programmers in this new world of .NET.



http://localhost/aspui.aspx

ASP.NET with Delphi

test

test

Document: Done (0.16 secs)

FIGURE 25.4 The output of the `aspui.aspx` example, after typing in the edit box and clicking the button

What's Next?

While you're waiting for the Delphi for .NET product, currently code-named Galileo, you can begin experimenting with the Delphi for .NET Preview that ships with Delphi 7 (and the subsequent updates made available by Borland). Of course, you should stay tuned to Borland's Developer Network website (`bdn.borland.com`) and newsgroups and to the author's site for update information about this area of Delphi, which is definitely a work in progress.

Just as Borland wants to provide the best tools to developers, I hope this book has helped you master Delphi, the most successful tool Borland has brought to the market in the last few years. Remember to check from time to time the reference, foundations, and advanced material I've collected on my website (`www.marcocantu.com`). Much of this material could not be included in this book, because of space constraints; see Appendix C, "Free Companion Books on Delphi," for more information.

Appendixes A and B discuss some of the add-ins I've built, which are freely available on my site, and a few other notable free Delphi tools. Also check my site for updates and integration of the material in the book, and feel free to use the newsgroups hosted there for your questions about the book and about Delphi in general.

Appendix A

Extra Delphi Tools by the Author

DURING THE LAST FEW years I have developed a number of small components and Delphi add-in tools. Some of these tools were written for books or were the result of extending examples from books. Others were written as a helper for repetitive operations. All these tools are available for free, and some include the source code. This appendix provides a list, including in particular tools mentioned in the book. Support for all these tools is available on my newsgroups (see `www.marcocantu.com` for directions).

CanTools Wizards

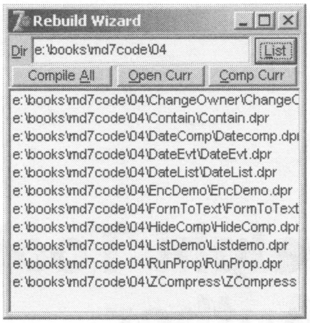
This is a set of wizards you can install in Delphi, either within an extra pull-down menu or as a submenu of the Tools menu). The wizards (freely available at `www.marcocantu.com/cantoolsw`) are unrelated and offer disparate features:

List Template Wizard Streamlines the development of similar list-based classes, each with its own type of objects. This wizard is mentioned in Chapter 4. Because it does a search-and-replace operation on a base source file, it can be used any time you need repeated code and the name of a class (or other entity) varies.

OOP Form Wizard (Mentioned in Chapter 4.) Allows you to hide the published components of a form, making your form more object-oriented and providing a better encapsulation mechanism. Start it when a form is active, and it will fill the `OnCreate` event handler. Then you must manually move part of the code into the unit initialization section.

Object Inspector Font Wizard Lets you change the font of the Object Inspector (something particularly useful for presentations, because the Object Inspector's font is too small to be seen easily on a projection screen). Another option available in the settings dialog allows you to toggle an internal feature of the Object Inspector and display the font names (in the drop-down combo box for that property) using a specific font.

Rebuild Wizard Allows you to rebuild all the Delphi projects in a given subfolder after loading each of them in sequence in the IDE. You can use this wizard to grab a series of projects (like those in a book) and open the one you are interested in by clicking on the list:



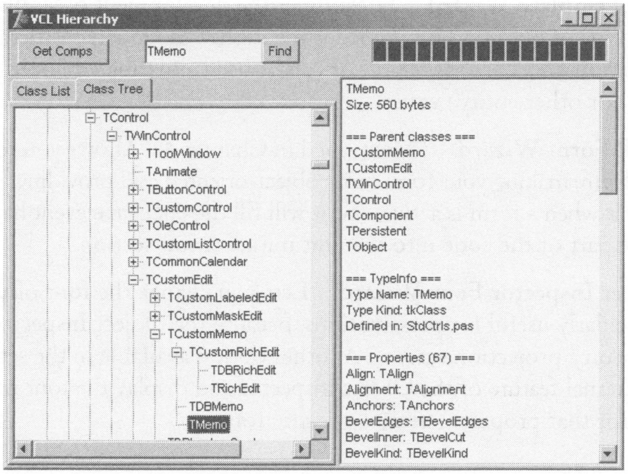
```
e:\books\md7code\04
```

```
e:\books\md7code\04\ChangeOwner\ChangeC
e:\books\md7code\04\Contain\Contain.dpr
e:\books\md7code\04\DateComp\Datecomp.dpı
e:\books\md7code\04\DateEvt\DateEvt.dpr
e:\books\md7code\04\DateList\DateList.dpr
e:\books\md7code\04\EncDemo\EncDemo.dpr
e:\books\md7code\04\FormToText\FormToText
e:\books\md7code\04\HideComp\HideComp.dpı
e:\books\md7code\04\ListDemo\Listdemo.dpr
e:\books\md7code\04\RunProp\RunProp.dpr
e:\books\md7code\04\ZCompress\ZCompress
```

You can also automatically compile a given project or start a (slow) multiproject build: In the compiler results dialog, you click a button to proceed only if the corresponding environment option is set. If this environment option is not set, you won't see the compiler errors, because the compiler messages are superceded at every compilation.

Clip History Viewer Keeps track of a list of text items you've copied to the Clipboard. A memo in the viewer's window shows the last 100 clipped lines. Editing the memo (and clicking Save) modifies this Clipboard history. If you keep Delphi open, the Clipboard will also get text from other programs (but only text, of course). I've seen occasional Clipboard-related error messages caused by this wizard.

VCL Hierarchy Wizard Shows the (almost) complete hierarchy of the VCL, including third-party components you've installed, and allows you to search one class and see many details (base and subclasses, published properties, and so on). Clicking the button regenerates both the list and the tree (in sequence—hence the progress bar runs twice):



```
TMemo
```

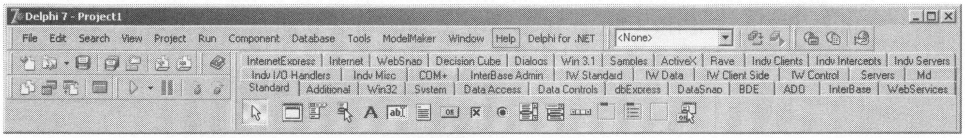
```
⊟ TControl
  ⊟ TWinControl
    ⊞ TToolWindow
      TAnimate
    ⊞ TButtonControl
    ⊞ TCustomControl
    ⊞ TOleControl
    ⊞ TCustomListControl
    ⊞ TCommonCalendar
    ⊟ TCustomEdit
      ⊞ TCustomLabeledEdit
      ⊞ TCustomMaskEdit
      ⊟ TCustomMemo
        ⊟ TCustomRichEdit
            TDBRichEdit
            TRichEdit
          TDBMemo
          TMemo
```

```
TMemo
Size: 560 bytes

=== Parent classes ===
TCustomMemo
TCustomEdit
TWinControl
TControl
TComponent
TPersistent
TObject

=== TypeInfo ===
Type Name: TMemo
Type Kind: tkClass
Defined in: StdCtrls.pas

=== Properties (67) ===
Align: TAlign
Alignment: TAlignment
Anchors: TAnchors
BevelEdges: TBevelEdges
BevelInner: TBevelCut
BevelKind: TBevelKind
```

The list of classes is generated by using predefined core classes (some are still missing; feel free to send suggestions) and then adding each component of the installed packages (Delphi's, yours, third parties'…) along with the classes of all the published properties having a class type. However, classes used only as public properties are not included.

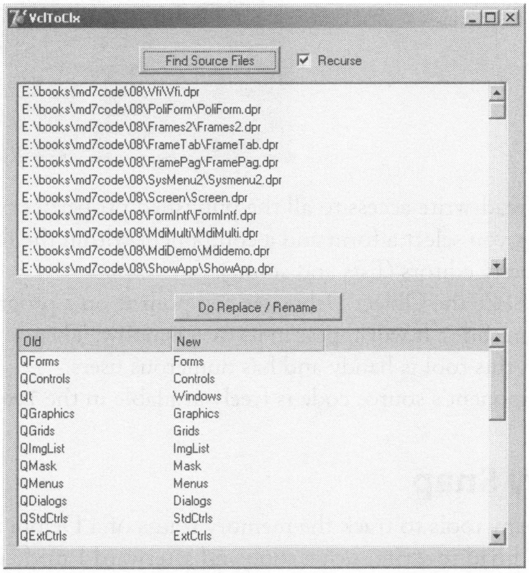
Extended Database Forms Wizard Does much more than the Database Forms Wizard available in the Delphi IDE, allowing you to choose the fields to place on a form and also to use a dataset other than those based on the BDE.

Multiline Palette Manager Allows you to turn Delphi's Component Palette into a TabControl with multiple lines of tabs:


Delphi 7 - Project1
File Edit Search View Project Run Component Database Tools ModelMaker Window Help Delphi for .NET
<None>
InternetExpress | Internet | WebSnap | Decision Cube | Dialogs | Win 3.1 | Samples | ActiveX | Rave | Indy Clients | Indy Intercepts | Indy Servers
Indy I/O Handlers | Indy Misc | COM+ | InterBase Admin | IW Standard | IW Data | IW Client Side | IW Control | Servers | Md
Standard | Additional | Win32 | System | Data Access | Data Controls | dbExpress | DataSnap | BDE | ADO | InterBase | WebServices

VclToClx Conversion Program

You can use this stand-alone tool to convert a Delphi project from VCL to CLX (and vice versa, if you configure it to do so). It can simultaneously convert all the files in a given folder and its subfolders. Here's an example of its output:


VclToClx
Find Source Files
☑ Recurse
Do Replace / Rename

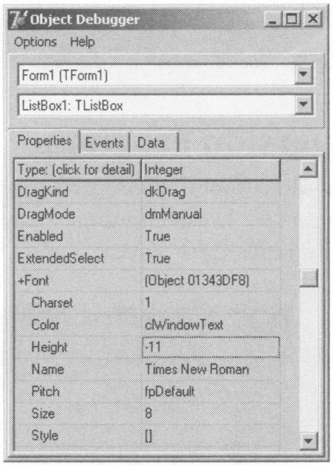
```
E:\books\md7code\08\Vfi\Vfi.dpr
E:\books\md7code\08\PoliForm\PoliForm.dpr
E:\books\md7code\08\Frames2\Frames2.dpr
E:\books\md7code\08\FrameTab\FrameTab.dpr
E:\books\md7code\08\FramePag\FramePag.dpr
E:\books\md7code\08\SysMenu2\Sysmenu2.dpr
E:\books\md7code\08\Screen\Screen.dpr
E:\books\md7code\08\FormIntf\FormIntf.dpr
E:\books\md7code\08\MdiMulti\MdiMulti.dpr
E:\books\md7code\08\MdiDemo\Mdidemo.dpr
E:\books\md7code\08\ShowApp\ShowApp.dpr
```

Old	New
QForms	Forms
QControls	Controls
Qt	Windows
QGraphics	Graphics
QGrids	Grids
QImgList	ImgList
QMask	Mask
QMenus	Menus
QDialogs	Dialogs
QStdCtrls	StdCtrls
QExtCtrls	ExtCtrls

The program's source code is available in the `Tools` folder of the book's source code. The VclToClx program converts unit names (based on a configuration file) and handles the DFM issue by renaming the DFM files to XFM and fixing the references in the source code. The program is unsophisticated—it doesn't parse the source code but instead looks for occurrences of the unit names followed by a comma or semicolon, as happens in a `uses` statement. It also requires the unit name to be preceded by a space, but you can modify the program to look for a comma. Don't skip this extra test; otherwise the Forms unit will be converted to QForms, but the QForms unit will be re-converted to QQForms!

Object Debugger

At design time, you can use the Object Inspector to set the properties of the components of your forms and other designers. In Delphi 4, Borland introduced a run-time Debug Inspector, which has a similar interface and shows similar information. Before Borland added this feature, I implemented a run-time clone of the Object Inspector meant for debugging programs:



Form1 (TForm1)

ListBox1: TListBox

Type: (click for detail)	Integer
DragKind	dkDrag
DragMode	dmManual
Enabled	True
ExtendedSelect	True
+Font	(Object 01343DF8)
Charset	1
Color	clWindowText
Height	-11
Name	Times New Roman
Pitch	fpDefault
Size	8
Style	[]

It allows read-write access to all the published properties of a component, and has two combo boxes that let you select a form and a component within the form. Some of the property types have custom property editors (lists and so on).

You can place the Object Debugger component on a program's main form (or you can create it dynamically in code): It will appear in its own window. There is room for improvement, but even in its current form this tool is handy and has numerous users.

This component's source code is freely available in the `Tools` folder of the book's source code.

Memory Snap

There are many tools to track the memory status of a Delphi application. During the development of a project, I had to write such a tool, and afterward I made it available.

I've written a custom memory manager that plugs into Delphi's default memory manager, keeping track of all memory allocations and de-allocations. In addition to reporting the total number (something Delphi also does now by default), it can save a detailed description of the memory status to a file.

Memory Snap keeps in memory a list of allocated blocks (up to a given total number, which can be easily varied), so that it can dump the contents of the heap to a file with a low-level perspective. This list is generated by examining each memory block and determining its nature with empirical techniques you can see in the source code (although they are not easy to understand). The output is saved in a file, because this is the only activity that doesn't require a memory allocation that would affect the results. Here is a snippet of a sample file:

```
157) 00C035CC:  object: [TList - 16]
158) 00C035E0:  buffer with heap pointer [00C032B0]
159) 00C03730:  string: [5-1]: Edit1
160) 00C03744:  object: [TEdit - 544]
161) 00C03968:  object: [TFont - 36]
162) 00C03990:  object: [TSizeConstraints - 32]
163) 00C039B4:  object: [TBrush - 24]
164) 00C039F4:  buffer with heap pointer [00C01FE4]
165) 00C03B34:  buffer with heap pointer [00C01F18]
166) 00C03B48:  string: [0-0]: dD
167) 00C03B58:  string: [11-2]: c:\mman.log
```

The program could be extended to profile memory usage by type (strings, objects, other blocks), track unreleased blocks, and keep memory allocations under control.

Again, the source code of this component is freely available in the `Tools` folder of the book's source code.

Licensing and Contributions

As you have seen, some of these tools are available with full source code. They are licensed under the LGPL (Lesser General Public License, `www.gnu.org/copyleft/lesser.html`), which means you can freely use and redistribute them in any way, including making modifications, while the author retains the original copyright. The LGPL doesn't allow you to close-source your extensions, but you can use this library code within programs you sell, regardless of the source code availability. If you extend these tools by fixing bugs or adding new features, I ask you to send your updates to me so I can further distribute the enhancements and we can avoid forking the code into too many versions. The license does not require you to do so, though.

Appendix B

Extra Delphi Tools from Other Sources

THOUSANDS OF DELPHI ADD-ON components and tools are available on the market, ranging from simple freebies to large open-source projects, from shareware programs to highly professional components. This appendix provides a list of notable open-source projects I've mentioned elsewhere in the book.

Delphi Preinstalled Open-Source Components

Delphi 7 includes the source code for two notable open-source projects:

Internet Direct (Indy) This project is covered in detail in Chapter 19, "Internet Programming: Sockets and Indy." The official website for Indy is `www.nevrona.com/indy`, but you should also refer to the Indy Portal at `www.atozedsoftware.com/indy`. Support is available on Borland's newsgroups.

Open XML This is a Delphi-based XML DOM and SAX engine, which I covered in Chapter 22 ("Using XML Technologies"). Refer to `www.philo.de/xml` for updates and more information. Support for Open XML is available on a mailing list you can sign up for on the website.

Delphi also includes a third open-source project: the ZLib compression library, covered in Chapter 4 ("Core Library Classes"). It isn't listed here because it is not Delphi based.

Other Open-Source Projects

The Delphi programmers' community has been very active since its inception and has produced numerous tools that were and are freely distributed among programmers. A subset of these tools is available with complete source code, as is often true for commercial Delphi components as well.

Project JEDI

The Joint Endeavor of Delphi Innovators, better known as Project JEDI (`www.delphi-jedi.org`), is not a single project but rather is the largest community of open-source Delphi developers. The site hosts its own projects, plus others contributed and maintained by members on separate websites.

Project JEDI started as an effort to translate APIs for specific Windows libraries distributed by Microsoft or other companies. Making available the Delphi units with the declarations of those APIs allowed any Delphi developer to use them easily. More recently, the goals of Project JEDI have been extended with the definition of many subprojects and groups. In addition to an ever-growing API library, you can find projects including the JEDI Visual Component Library (JVCL), the JEDI Code Library (JCL, a set of utility functions and non-visual classes, including a nice un-handled exception stack tracer), plus many projects in the area of graphics, multimedia, and game programming. Other activities range from a JEDI Version Control System client to the DARTH header conversion kit, from a programmer's editor to online tutorials.

Other JEDI-associated projects include Indy (covered earlier), GExperts, and the Delphree site (covered next).

GExperts

GExperts (`www.gexperts.org`) is probably the most widespread add-in for the Delphi IDE, providing features ranging from a multiline Component Palette to source code navigation aids. Self-described as "a set of tools built to increase the productivity of Delphi and C++Builder programmers," it includes a large collection of wizards, including Procedure List, Clipboard History, Expert Manager, Grep Search, Grep Regular Expressions, Grep Results, Message Dialog, Backup Project, Set Tab Order, Clean Directories, Favorite Files, Class Browser, Source Export, Code Librarian, ASCII Chart, PE Information, Replace Components, Component Grid, IDE Menu Shortcuts, Project Dependencies, Perfect Layout, To Do List, Code Proofreader, Project Option Sets, and Components to Code.

The Delphree Site

There are many other Delphi-related projects in the open-source and community development areas. Although I cannot list all the relevant ones here, I can refer you to a website that tries to track all such projects. The Delphree (Delphi Free) website is available at `delphree.clexpert.com`.

This site has a comprehensive list of Delphi open-source projects, ranging from libraries to programmers' tools to end user applications.

DUnit

Among the many ideas promoted by extreme programming, I find unit testing particularly interesting. *Unit Testing* is the continuous development of test code even before a program is written. To do this, it is important to have a proper test framework. A group of Delphi programmers created such an architecture, called DUnit. You can find it on SourceForge at `dunit.sourceforge.net`.

Appendix C

Free Companion Books on Delphi

THIS IS THE SEVENTH edition of *Mastering Delphi*, and I've also written other books, material for classes and conference presentations, and much more; so, in addition to the thousand or so pages this book can accommodate, I have a lot of other material about Delphi programming, particularly introductory material. Over the last few years, I've turned this information into electronic books (in HTML or PDF format) available for free on my website. This appendix outlines the titles and their contents, and provides references to the download pages (all hosted on `www.marcocantu.com`).

Essential Pascal

Essential Pascal is an introduction to the foundations of the language invented by Professor Nicklaus Wirth and spread by Borland with its world-famous Turbo Pascal compilers during the 1980s. The 100-page book doesn't cover the OOP extensions, but details many features Borland added to the core language over the years. The following is a list of this e-book's chapters, available at `www.marcocantu.com/epascal`:

Chapter 1: "Pascal History"
Chapter 2: "Coding in Pascal"
Chapter 3: "Types, Variables, and Constants"
Chapter 4: "User-Defined Data Types"
Chapter 5: "Statements"
Chapter 6: "Procedures and Functions"
Chapter 7: "Handling Strings"
Chapter 8: "Memory (and Dynamic Arrays)"
Chapter 9: "Windows Programming"
Chapter 10: "Variants"
Chapter 11: "Programs and Units"

Chapter 12: "Files in the Pascal Language"
Appendix A: "Glossary of Terms"
Appendix B: "Examples"

As you can see on the website, the book has been translated by volunteers into quite a few languages.

Essential Delphi

Mastering Delphi has changed from an introductory text to an intermediate one as I've tried to cover most of the features Borland has added to Delphi over the years. In the process, I had to cut some of the material aimed at programmers who've never used Delphi or a visual tool. This material is now contained in the *Essential Delphi* e-book, freely available in PDF format on my site at `www.marcocantu.com/edelphi`. Here is the table of contents (which is still under development):

Chapter 1: "A Form Is a Window"
Chapter 2: "Highlights of the Delphi Environment"
Chapter 3: "The Object Repository and the Delphi Wizards"
Chapter 4: "A Tour of the Basic Components"
Chapter 5: "Creating and Handing Menus"
Chapter 6: "Multimedia Fun"
Chapter 7: "Saving Settings: From INI Files to the Registry"
Chapter 8: "More on Forms"
Chapter 9: "Delphi Database 101 (with Paradox)"
Chapter 10: "Printing"
Appendix A: "Essential SQL"
Appendix B: "Common VCL Properties"

Delphi Power Book

Finally, I'm making available more material that covers advanced topics, in a collection called *Delphi Power Book*. At the time of this writing, only four chapters of this e-book are available:

"Graphics in Delphi", "Examples of Using Interfaces", "COM Shell Extensions", and "Debugging."

A lot more will come; stay tuned to the website `www.marcocantu.com/delphipowerbook`.

Index

Note to the Reader: Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

A

- About boxes, **289–292**, *290*
- Absolute function, 921
- abstract classes, 69, 338
- abstract methods, **66–67**
- AbstractAnimals example, 67
- acBoldUpdate method, 229
- accelerator keys, **181–182**
- Access databases, 624
- access rights in WebSnap, **808**
- access specifiers, **48–50**, **925**
- acCountcharsUpdate method, 229
- acCutUpdate method, 229
- AcManTest example, 244–247, *244*
- acPasteUpdate method, 229–230
- acSaveUpdate method, 229
- action objects, 222
- Action parameter in close, 279
- Action property, 223
- ActionBars collection, 244, 246
- ActionBoldExecute method, 227
- ActionCount object, 227
- ActionCountExecute method, 227
- ActionEnableExecute method, 227
- ActionFont object, 313
- ActionFontExecute method, 313
- ActionFontUpdate method, 314
- ActionIncreaseExecute method, 539
- ActionLink object, 223
- ActionList component, 173, **222–224**, *223*
  - example, **225–228**, *226*, *228*
  - predefined actions in, **224–225**
  - with toolbars, **229–230**
- ActionList editor, 226, *226*
- ActionListUpdate method, 230
- ActionMainMenuBar control, 242
- ActionManager architecture, **241–242**
  - example, **242–245**, *243–244*
  - for least-recently used menu items, **245–247**, *245*
  - for list actions, **248–250**, *249*
  - for porting programs, **247–248**
- ActionManager component, 242
- ActionNewExecute method, 314
- ActionOpenExecute method, 314
- actions, **173**
  - ActionList for, **222–224**, *223*
  - defining, **385–388**
  - example, **225–228**, *226*, *228*
  - list, **248–250**, *249*
  - predefined, **224–225**
  - for toolbars, **229–230**
  - for transactions, 596
- Actions example, 225–228, *226*, *228*
- Actions property, 771–772
- ActionSaveAsExecute method, 314
- ActionSaveExecute method, 315
- ActionSaveUpdate method, 314
- ActionShowStatusExecute method, 246
- ActionToolBar control, 242
- ActionTotalExecute method, 539
- ActivApp example, 298, *298*
- activating
  - applications and forms, **298**, *298*
  - controls, **159**
- activation messages, 374
- active documents, 456
- Active property, 350, 509
- ActiveButton component, **372–373**, *373*
- ActiveChange method, 676
- ActiveChanged method, 678–679, 683
- ActiveForm property, 299
- ActiveForm Wizard dialog box, 491
- ActiveForms, **491–492**
- ActiveMDIChild property, 312
- ActiveScripting engine, 791
- ActiveX Control Wizard, 487–488, *487*
- ActiveX controls, 456, **483–484**
  - ActiveForms, **491–492**
  - arrow, **487–488**, *487*
  - vs. components, **484–485**
  - properties for, **488–489**, *488*
  - property pages for, **489–491**, *490*

  - in Web pages, **492–493**, *493*
  - WebBrowser control, **485–486**, *486*
  - XClock, **492–493**, *493*
- ActiveX Data Objects. *See* ADO (ActiveX Data Objects)
- ActiveX libraries, 34, 460–461, 488
- ActiveX page, 461, 487, 489
- activity diagrams, 434
- actors in use case diagrams, 434
- AdapterCommandGroup, 795
- AdapterDispatcher component, 787
- AdapterErrorList component, 803
- AdapterFieldGroup component, 795
- AdapterGrid component, 799
- AdapterMode property, 801, 803
- AdapterPageProducer component, **795–797**, *796*
  - for formview page, 802–803
  - for logins, 807
  - for main page, 798–801
  - for master/detail relationships, 803
- adapters, **794**
  - AdapterPageProducer for, **795–797**, *796*
  - components for, **794–795**
  - fields for, **794**
  - for locating files, **798**
  - scripts for, **797–798**
- adCriteria constants, 640
- Add Field command, 531
- Add Fields dialog box, 526, *526*
- Add Files to Model option, 437
- Add method
  - for collections, 382
  - in Delphi for .NET Preview, 926–927
  - in TBucketList, 132
  - in TList, 133
  - in TMdObjDataSet, 713
  - in TreeView, 194
  - in TStringList and TStrings, 128
- Add To-Do Item command, 9
- Add To Interface dialog box, 488–489, *488*
- Add to Model option, 437
- Add To Repository command, 40
- AddChild method, 194
- AddChild node, 844
- AddColors method, 187–188
- AddDefaultCommands property, 796
- AddDefaultFields property, 796
- AddFilesToList function, 103
- Additional page, 180
- AddNode method, 197
- AddObject method, 187
- Addr function, 921
- AddRecord method, 701
- _AddRef method, 70, 457
- addresses
  - of functions, 63
  - of methods, 115
  - in socket programming, **739**
- AddToList method, 751, 753
- AddWebModuleFactory method, 792
- AdjustLineBreaks function, 87
- administrative components in IBX, **593**
- ADO (ActiveX Data Objects), **508**, **615–616**
  - briefcase model in, **645**
  - client indexes in, **632**
  - cloning in, **633**, *633*
  - connection pooling in, **643–644**
  - cursors in, **629–632**
  - Data Link files for, **621**
  - dbGo for, **618–620**, *619–620*
  - dynamic properties in, **622**
  - Jet engine for, **624–629**, *625–626*
  - locks in, **635–636**, **639–641**
  - in MDAC, **616–617**
  - OLE DB providers, **616–617**
  - recordsets in
    - disconnected, **642–643**
    - persistent, **644–645**
  - schema information in, **622–623**, *623*
  - transaction processing in, **634–636**
  - updates in, 615
    - batch, **638–642**
    - joins, **636–637**
- ADO Cursor Engine, 629
- ADO Multi-Dimensional (ADOMD), 617
- ADO.NET, **645–646**
- ADOCommand component, 508, 618
- ADOConnection component, 508, 618, **621**, 628, 634–635
- ADODataSet component, 508, 618–619, 626
- ADOExpress, 616
- ADOMD (ADO Multi-Dimensional), 617
- ADOQuery component, 508, 618–619, 626
- ADOStoredProc component, 508, 618–619
- ADOTable component, 508, 618–619, 621, 624, 626
- ADOUpdatesPending function, 638
- ADOX technology, 623

ADTG (Advanced Data Table Gram) format, 644–645
Advanced page, 619
AfterEdit event, 524
AfterInsert event, 547
aggregates, **550–552**, *550*
Aggregates property, 550
aggregation of interfaces, 921
Align property, 181
Alignment dialog box, 19
Alignment property, 528, 658
Alignment toolbar, 718
All page, 619, 624
AllocMemCount variable, 84, 335
AllocMemSize variable, 84, 335
AllocRecordBuffer method, 699
AllowAllUp property, 230
AllowGrayed property, 165–166
AllowHiding property, 244
alpha blending, **267**, *267*
AlphaBlend function, 267
AlphaBlendValue property, 267–268
AlphaDIBBlend function, 267
alter procedure statement, 565
alter trigger statement, 566
alternatives in XSLT, 864
ampersands (&) for accelerator keys, 181
ancestor classes, 63
AncestorIsValid function, 148
Anchors example, 179–180, *179*
anchors for controls, **179–180**, *179*
Anchors property, 179
angle brackets (<>) in XML, 834
animated displays, **268**
AnimateWindow API, 268
annotations on diagrams, 435
AnsiContainsText function, 91
AnsiDequotedStr function, 87
AnsiIndexText function, 91–92
AnsiMatchText function, 91–92
AnsiQuoteStr function, 87
AnsiReplaceText function, 91
AnsiResembleText function, 91
Apache Web Server, 767, **781–872**
apartment threading model, 463
AppendRecord method, 701
AppID property, 817
Application Form option, 820
application-level interfaces, 615
Application mode in IntraWeb, 815
Application object, 183, 219, **295–297**, 793
    for activating applications and forms, **298**, *298*
    application window in, **297–298**
    with DLLs, 411
    from TCGIApplication, 771
    for tracking forms, **299–302**, *300*
Application page, 295
application window, **297–298**
Application Wizard, 825
ApplicationAdapter component, 788
ApplicationEvents component, 296
applications
    activating, **298**, *298*
    MDI. *See* MDI (Multiple Document Interface) applications
    messages for, 374
    titles for, 297
    upsizing, 556
    Web. *See* IntraWeb library
Applications page, 28
ApplicationTitle property, 788
ApplyBitBtnClick method, 287
ApplyUpdates method, 575, 586, 589, 660, 803
AppServ2 example, 657–658
AppServer property, 664
AppSPlus example, 662–664, 668
Architect Studio edition, 4
Arrange option, 21
ArrangeIons method, 312
array-based properties, 52
arrays in ModelMaker, 450
Arrow component, **358**
    ActiveX, **487–488**, *487*
    enumerated properties for, **358–360**
    events for, **364–366**
    paint method for, **361–362**, *362*
    persistent properties for, **362–364**, *364*
    registering property categories for, **366–367**, *367*
arrow keys for moving components, 18
ArrowDblClick method, 365
ArrowHeight property, 359, 489
AS methods in IAppServer, 650
as operator, 68, 70, 465
As properties, 528
ASCII text files, source files as, 38
AsDateTime property, 574

- ASM statements, 921
- ASP.NET, **940–942**, *942–943*
- aspbase.aspx example, 941–942, *942*
- aspui.aspx example, 942, *943*
- assemblies
  - in .NET architecture, **906–907**
  - type of, 916
- Assign method, 57–58
  - in TClipboard and TBitmap, 208
  - in TList, 128
  - in TPersistent, 113
- assigning objects, **56–58**, *56*
- AssignPrn method, 582
- AssignTo method, 113
- Associate property, 172, 275
- association symbols in diagrams, 435
- associative lists, **132–133**
- AsSQLTimeStamp property, 574
- AsText method, 495
- asymmetric arithmetic rounding, 90
- atomic operations, 589
- AttachedMenu property, 816
- attachments, **892–893**, *894*
- Attribute class, 928
- attributes
  - in Delphi for .NET Preview, **927–928**
  - in XML, 835
  - in XSLT, 865
- Attributes property, 635
- AutoActivate property, 482
- AutoCheck property, 173, 222
- AutoComplete property, 168
- AutoConnect property, 477
- AutoDock property, 235
- AutoDrag property, 235
- AutoHotkeys property, 182
- automated keyword, 921
- automatic operations
  - form scaling, **276–277**
  - form scrolling, **272–273**
  - mail message generation, 748
- Automation, 456, **467–468**
  - call dispatching in, **468–470**, *469*
  - clients in, **474–475**
  - compound documents, **479–483**, *482*
  - internal objects in, **482–483**, *483*
  - object scope in, **476–477**
  - Office programs with, **478–479**
  - servers, **470**, *470*
    - code for, **472–473**
    - in components, **477–478**, *477*
    - registering, **473–474**
    - type-library editor for, **470–472**, *471*
  - speed differences in, **476**
  - types in, **478**
- Automation Object Wizard, 470, *470*
- AutoSave feature, 11
- AutoScroll property, 270, 272, 275, 277
- AutoSize property, 234–235
- AutoSnap property, 181
- availability in InterBase, 561
- AxBorderStyle property, 492
- AxCtrls unit, 478

B

- B2B (business-to-business) applications, 875
- BabelFish translations, **876–879**, *877*, *879*
- background processing, **304**
- BackPages object, 212
- BackTask example, 305–307
- band components, **726–728**, *727*
- Band Style Editor, **727–728**, *727*
- Bands property, 231
- BandStyle property, 726
- banker's rounding, 90
- bar code components, **725**
- BarMenuClick method, 234
- base classes, **330–333**, **340**
- base forms
  - inheritance from, **319–321**, *319–320*
  - and interfaces, **330**, **333–334**
- base types, 84
- BaseCLX, 110–111
- batch updates, 615, **638–642**
- BDE (Borland Database Engine), **507**, 615
- BeforeDestruction method, 279
- BeforeEdit event, 524
- BeforePost event, 546
- beginning-of-file cracks, 697
- BeginThread function, 84
- BeginTrans method, 634
- Beveled property, 180
- Beyond Compare utility, 11
- bi constants, 256–257
- BIcons example, 256–257, *257*

binary large object (BLOB) fields
  in DataSnap, 666
  in record viewer, 682
  streams for, 136, 139
binding
  late. *See* late binding
  resource files into executables, 30
binding interfaces in XML, **846–850**, *847*
Bitmap component, **724**
bitmaps
  for Component Palette, **348**, *349*
  for components, 340, 348
  for ListView, 188
  for Rave, **724**
  for TabControl, 209
  viewer for, 207–210, *207*
black box programming, 48
BLOB (binary large object) fields
  in DataSnap, 666
  in record viewer, 682
  streams for, 136, 139
blocking sockets, **738**
BlockRead function, 921
BlockWrite function, 921
BMP files, 34. *See also* bitmaps
BmpSide constant, 210
BmpViewer example, 207–210, *207*
Bof tests, 537
BofCracks, 697
BoldExecute method, 215
booking classes, **608–611**, *610*
Bookmark property, 538, 542
bookmarks, **537–538**, 633, **695–698**, *696*
  shortcut keys for, 16
  in TDataSet, 687
Boolean conversions, 86
BoolToStr function, 86
border icons, **256–257**, *257*
BorderIcons property, 255–256
Borders example, 254–256, *254*
borders for forms, **254–256**, *254*
BorderStyle property, 254–255
Borland Database Engine, **507**, 615
Borland Registry Cleanup Utility, 33
Borland resource compiler tool, 33
BorlndMM.DLL file, 405–406
BoundsRect method, 159
BPG files, 28, 34
BPL files, 34, 338, 347
BPL folder, 347
BRC32.exe and BRCC32.exe tools, 33
Break property, 174
briefcase model, **645**
Bring To Front command, 204
BringToFront method, 302, 328
BrokDemo example, 774–778, *777*
BrowseFast example, 756–757, *757*
browsers
  creating, **756–757**, *757*
  in UDDI, **895–898**, *896*
browsing in editor, **12–13**
bs constants, 255
btnSendClick method, 745
BtnTodayClick method, 56–58
BucketCount property, 133
BucketFor function, 133
buffers and caches
  with ClientDataSet, **584**
  for dataset records, **698–703**
  in TDataSet, 687
Build Later command, 27
Build Sooner command, 27
building projects, **30–32**
Bushakra, John, 899
business rules, 648, 656–657, 711
business-to-business (B2B) applications, 875
BusinessEntity type, 895
BusinessService type, 895
BusinessTemplate type, 895
button wheels, 261
ButtonLoadClick method, 323
ButtonStyle property, 516, 533
Byte type, 478

C

C# programming language, 10
C++ programming language, **401–402**
CAB files, 34, 493
cached updates, **638–642**
CacheFile property, 893
caches and buffers
  with ClientDataSet, **584**
  for dataset records, **698–703**
  in TDataSet, 687
CacheSize property, 630–631
caFree value, 279

- caHide value, 279
- Calc example, 531–533, *533*
- CalcController component, **733**
- CalcOp component, **733–734**
- CalcText component, **729–730**
- CalcTotal component, **733**
- CalcType property, 730
- calculated fields
  - adding, **530–533**, *531*, *533*
  - internally calculated, 513
- calculations in Rave, **733–734**
- callback functions, 308
- callback interfaces, 497–498
- callback mechanisms, 910
- CallFirst example, 406, *407*
- calling
  - DLLs, **406**, *407*, **408–410**
  - procedures, 409–410
- caMinimize value, 279
- CancelBatch method, 638
- CancelUpdates method, 638
- caNone value, 279
- CanTools wizards, **945–947**
- Canvas property, 262
- CaptionPlus method, 57
- captions and Caption property
  - for actions, 222
  - inputting, 260
  - for menu items, 174
  - for property pages, 490
  - removing, 258
  - in TabSheet, 203
  - toolbar, 236
- Capture property, 264–265
- Cascade method, 312
- Cascading Style Sheets (CSS), **764–765**, 797, 859
- case-insensitive searches, **602–603**
- case toggling, shortcut keys for, 16
- casting
  - RTTI for, **67–69**
  - type-safe down-casting, **67–69**
- categories, property
  - in Object Inspector, **21–22**
  - registering, **366–367**, *367*
- Category property, 222, 224
- CCLibDir entry, 26
- CDATA section, 834
- CDK (Component Development Kit), 340
- CDS formats, **511**, *512*
- cdsCalcFields method, 531–532
- CdsCalcs example, 549–553, *550*
- CdsDelta example, 586–588, *586*
- CdsXstl example, 868–869
- CFG files, 34
- CGI programs, **768–769**, *769*
- CgiDate example, 768–769, *769*
- CgiIntra example, 823–824
- ChangeAlignment method, 230
- ChangeColor method, 476
- Changed method, 344, 383
- ChangedAllowed property, 244
- ChangeFamily method, 97
- ChangeFormFont property, 344, 348
- ChangeOwner example, 119–120, *120*
- ChangesToClone property, 712
- character sets in XML, 835
- characters event, 850, 852
- characters method, 872
- CheckBiDirectional function, 572
- CheckBox component, **165–166**
- CheckBoxList component, **169**
- CheckCapsLock method, 218
- Checked property
  - for actions, 222, 230
  - in CheckBoxList, 169
- child components, messages for, 374
- child forms in MDI applications, **315–316**, *315*
- child windows in MDI applications, **311–315**, *313*
- CIL (Common Intermediate Language), 904, **907–908**
  - CTS in, **909–911**
  - managed and safe code in, **908–909**
- class completion
  - in editor, **13–14**
  - for methods, 45
- Class_DColorPropPage constant, 489
- Class_DFontPropPage constant, 489
- Class_DPicturePropPage constant, 489
- Class_DStringPropPage constant, 489
- class factories, **460**
- class helpers, **929–931**
- class interfaces, 48
- class references, **76–78**, *78*
- classes, 109. *See also* components
  - in Delphi, **44–48**
  - diagrams for, **431–432**, *432–433*
  - for exceptions, **73–75**
  - for fields, **529–530**
  - information for, **104–108**, *108*

    interposer, **332–333**
    moving, 446
    in Project Explorer, 32
    reparenting, 446
Classes unit, **147–148**
ClassHelperDemo example, 929
ClassInfo example, 107–108, *108*
ClassInfo method, 106
ClassName method, 104–105
ClassParent method, 105
ClassRef example, 77–78, *78*
ClassType method, 105
clear method for dialog boxes, 283–284
CLI (Common Language Infrastructure), **903–904**
client areas
    in controls, 159
    in forms, **269**
    for mouse events, 262
client cursors, 629
client indexes in ADO, **632**
client/server programming, **555–557**
    with ClientDataSet
        aggregates in, **550–552**, *550*
        grouping in, **549**, *550*
        packets and caches in, **584**
        transactions in, **589–591**, *591*
        updates in, **584–589**, *585–586*, *588*
    database design for, **557–561**
    with dbExpress library, **566**
        components, **569–574**, *571*, *573*
        metadata in, **578–579**, *579*
        parametric queries in, **579–581**, *580–581*
        platforms and databases, **567–569**
        printing in, **581–584**
        single and multiple components with, **575–578**, *577*
        unidirectional cursors, **566–567**
    with InterBase, **561–566**, *563–564*
    unidirectional cursors in, **560–561**
client socket connections, 740
client windows, subclassing, **317–318**
ClientDataSet component, **508–509**, 567, 575
    client/server programming with
        aggregates in, **550–552**, *550*
        grouping in, **549**, *550*
        packets and caches in, **584**
        transactions in, **589–591**, *591*
        updates in, **584–589**, *585–586*, *588*
    for DataSnap, 649, 655–656, *656*
    filtering for, **513**
    formats in, **511**
    indexing for, **512–513**
    local tables for
        connections for, **509–510**, *510*
        defining, **511–512**, *512*
    locating records in, **514–515**
    master/detail structures in, **552–553**, *553*
    Midas library for, **510–511**
    for updates, 576
    in WebSearch program, 785
    in WebSnap, 799, 803
    in XML, 838, 854–856, **870–873**
ClientHandle property, 312
ClientHeight property, 269
ClientRefresh example, 661, *661*
clients
    Automation, **474–475**
    for DataSnap, **652–653**
    MDI, 310–311
    thin, 649
    in Web database applications, **831–832**, *832*
ClientToScreen method, 175, 183
ClientWidth property, 269
Clip History Viewer, 946
Clipboard
    for actions, 229–230
    for components, 24–25
    for DDE, 456
    with TabControl, 208–209
    for toolbar operations, 214–215
ClipBrd unit, 208
ClipCursor function, 263
clocks
    digital, 349–352
    XClock, **492–493**, *493*
Clone method, 633
cloneNode, 843
clones
    in ADO, **633**, *633*
    property, 187
    recordset, 633
Close method, 279–280
closing
    datasets, **694–695**
    forms, **279–280**
closing markers in XML, 834–835
CLR (Common Language Runtime), **905–906**
CLRReflection example, 934–940, *940*
CLS (Common Language Specification), **905–906**

- CLX (Component Library for Cross-Platform), 5, **110–112**
  - structure of, **111–112**
  - styles for, **219–220**, *220*
- cm_ prefix, 373–378
- cm_DialogChar message, 377
- cm_DialogKey message, 260, 377
- cm_FocusChanged message, 378
- cm_MouseEnter message, 372
- cm_MouseExit message, 372
- CmdGotoPage command, 799
- CmdLine global variable, 84
- CmdNextPage command, 799
- CmdPrevPage command, 799
- CmExit method, 677
- CMNTest example, 377–378
- cn_ prefix, 373, 377
- CNHScroll method, 676
- CNVScroll method, 676–677
- CoCreateGuid function, 459
- code completion, **14–15**
- Code Explorer window, **11–12**, *12*
- code generation vs. streaming, **113–114**
- code in Delphi for .NET Preview, **921–922**
- code insight technology, **14**
- code parameters, **15**
- Code Template Parameters dialog box, 450, *450*
- code templates, **15**
- CoFirstServer class, 475
- ColHeights property, 683
- collaboration diagrams, 434
- collections
  - classes in, **131**
  - for properties, **381–385**, *383*
- colons (:) in to-do items, 9
- color
  - for controls, **160–161**
  - converting to strings, 187–188
  - in DLLs, 404, 410–411
  - list box of, **186–188**
  - messages for, 377
  - in Object Inspector, 20
  - transparent, **267**, *267*
- Color property
  - in Splitter, 180
  - in TControl, 159–160
- ColorBox component, **169**
- ColorDialog component, 288
- ColorDrawItem method, 185–186
- ColorKeyHole example, 267–268, *267*
- ColorRef type, 404
- Colors toolbar in Rave, 718
- ColorToString function, 187–188
- Column property, 763
- ColumnLayout property, 167
- columns and Columns property
  - in DataSetTableProducer, 763
  - in DBGrid, 382, 516
  - editing, **539–540**
  - in ListBox, 167
  - in RadioGroup, 166
  - totals of, **537**, *537*
- Columns property editor, 533
- COM (Component Object Model), **455–456**
  - and ADO.NET, **645–646**
  - class factories in, **460**
  - data types in, **478**
  - in Delphi, **500–502**, *502*
  - GUIDs in, **458–460**, *459*
  - history of, **456–457**
  - instancing and threading methods in, **463**
  - Interop services in, 908
  - IUnknown for, **457–458**
  - server for, **460–461**
    - interface properties for, **465–466**
    - interfaces and objects for, **461–463**, *462*
    - object initialization for, **464**
    - testing, **464–466**
    - virtual methods in, **466–467**
- COM+, 456, **494**, 650
  - components for, **494–496**, *495–496*
  - events for, **497–500**, *499*
  - transactional data modules for, **497**
- COM Component Install Wizard, 498
- COM+ Event Object wizard, 498
- COM Object Wizard, 461–462, *462*, 498
- combo boxes, **216–217**, **341–348**, *345–346*
- ComboBox component, **168**
- ComboBoxEx component, **169**
- ComConts unit, 96
- ComDlg32.DLL file, 287
- CommandHandlers class, 741
- commands, **173**
  - and actions, **173**
  - Menu Designer for, **174**, *174*
  - pop-up menus, **174–176**
- CommandText Editor, 573, *573*

CommandText property
  in SQLDataSet, 573
  in TADODataSet, 626
CommandType property
  in SQLDataSet, 573
  in TADODataSet, 626
CommDlg example, 287–288, *288*
comments
  in ModelMaker, **444–445**, *444*
  TODO, **7–9**
  in XML, 834
Commit action, 596
Commit method, 589
CommitRetaining command, 596–597
CommitTrans method, 634
common dialog boxes, **287–288**, *288*
Common Intermediate Language (CIL), 904, **907–908**
  CTS in, **909–911**
  managed and safe code in, **908–909**
Common Language Infrastructure (CLI), **903–904**
Common Language Runtime (CLR), **905–906**
Common Language Specification (CLS), **905–906**
Common Object Request Broker Architecture (CORBA), 651
Common Type System (CTS), 903–904, **909**
  events and delegates in, **910–911**
  objects and properties in, **909–910**
ComObj unit, 96, 457
Comp type, 920
CompareValue function, 89
comparisons
  for ClientDataSet, 513
  with floating-point numbers, 89
  with strings, 91–92
compatibility
  in DLLs, 404–405
  with Linux, 88
  type, in inheritance, **62–63**
compilation
  conditional, **156**
  for libraries, **156**
  .NET Preview, 900
  for projects, **30–32**
compiler messages, **9–10**, 31, *31*
Compiler Messages page, 31, *31*
Compiler page, 28
complex numbers, 95
Component Development Kit (CDK), 340
component diagrams, 434
Component Library for Cross-Platform (CLX), 5, **110–112**
  structure of, **111–112**
  styles for, **219–220**, *220*
Component Object Model. *See* COM (Component Object Model)
Component Palette, **23–24**
  bitmaps for, **348**, *349*
  for copying and pasting components, **24–25**
  for databases, 507
  submenus on, 5
  templates and frames in, **25–26**, *26*
Component property, 395
component streaming system, 52
Component Template Information dialog box, 25
component templates, **25–26**
Component Wizard, 341–342
ComponentCount property, 119
ComponentIndex property, 118
components, 109, **162**. *See also* classes; controls
  vs. ActiveX controls, **484–485**
  for adapters, **794–795**
  Automation servers in, **477–478**, *477*
  base classes for, **340**
  class references for, **76–78**, *78*
  collection properties, **381–385**, *383*
  for COM+, **494–496**, *495–496*
  commands, **173–176**
  compound, **349**
    external components in, **352–354**, *353*
    frames for, **357–358**
    graphical. *See* Arrow component
    interfaces for, **354–356**
    internal components in, **349–350**
    publishing subcomponents of, **350–352**, *352*
  copying and pasting, **24–25**
  dynamic creation of, **47–48**, *47*
  editors for, **392–395**, *394*
  extending libraries for, **337–340**
  Fonts combo box, **341–348**, *341*, *345–346*
  hiding, 159
  input focus, **176–178**, *176–177*
  lists, **167–172**, *170*
  messages for, 376
  without names, 122
  nonvisual dialog, **378–381**, *381*
  for options, **165–166**
  packages for, **337–339**
  property editors, **388–392**, *390–391*

    ranges, **172–173**
    text-input, **162–165**, *165*
    windows controls. *See* windows
    writing, **339–340**
Components property, **118–120**
ComponentState property, 300
ComponentStyle property, 351
ComponentToDOM method, 845
compound components, **349**
    external components in, **352–354**, *353*
    frames for, **357–358**
    graphical. *See* Arrow component
    interfaces for, **354–356**
    internal components in, **349–350**
    publishing subcomponents of, **350–352**, *352*
compound documents, 456, **479–483**, *482*
compressed CAB files, 493
compressing streams, **145–146**, *145*
CompressStream function, 146
ComputePoints method, 360–361, 365
ComputerName property, 652, 655
ComServ unit, 96
concurrency
    access in, 557
    in COM+, 496
    InterBase for, 561
conditional compilation, **156**
conflicts in ADO, **641–642**
Connected property, 655
Connection component, 575
connection protocols for DataSnap, **650–651**
connection string editor, 618–619, *619–620*
connection strings, 618–619
ConnectionBroker component, 652–653, **666**
ConnectionName property, 571
connections
    ClientDataSet, **509–510**, *510*
    dbExpress, **576–577**, *577*
    pooling, **643–644**
    socket, **740**, **744–747**
connections.ini file, 570, 572
ConnectionString property, 618–619, 621, 629
ConnectKind property, 478
constant strings, 85
ConstraintErrorMessage property, 657
constraints and Constraints property, 179, 657
    in DataSnap example, **657–658**
    for forms, **270**
    in Splitter, 180
constructor keyword, 54
constructors, **54–55**, 926
container classes, **131–132**
containers, type-safe, **133–135**
contains keyword, 346
contains lists, 345–346
Contains section, 348
Content method
    in DataSetTableProducer, 762
    in PageProducer, 760
    in QueryTableProducer, 778
Content property, 772
ContentFields property, 778
ContentStream property, 784
ContentType property, 784
Contnrs unit, 131
Contrib.INI file, 692–693
ControlBar component, **231–234**, *232*
    docking toolbars in, **235–239**, *236*
    menus ins, **234**
ControlBarLowerDockOver method, 247–248
ControllerBand property, 727, 731
controllers in Automation, 467
controls, 111, **149**. *See also* components
    activation and visibility of, **159**
    ActiveX. *See* ActiveX controls
    anchors for, **179–180**, *179*
    colors for, **160–161**
    converting, **156–158**
    data-aware. *See* data-aware controls
    dual libraries support for, **151–155**, *152*
    fonts, **159–160**
    forms. *See* forms
    ListView, **188–193**, *191*
    menus. *See* menus
    owner-draw, **184–185**
        list box of colors, **186–188**
        menu items, **185–186**, *186*
    parents of, **158–159**
    size and position of, **159**
    status bars, **217–219**, *218*
    TControl for, **158–162**
    ToolBar, **213–219**, *214*
    TreeView, **193–197**, *194*, *199*
    TWidgetControl, **161–162**
    TWinControl, **161**
    VCL vs. VisualCLX, **149–158**, *152*
    windows. *See* windows
Controls property, 159

ConvDemo example, 97, *98*
conversions, **96–99**, *98*
  Booleans, 86
  colors, 187–188
  controls, **156–158**
  currency, **99–102**, *101*, **880–882**, *881*, *883*
  DFM files, 33, 153
  floating-point numbers, 87
  members, 447
  stream data, 140–142
  strings, 86–87, 187–188
  temperatures, 96
  types, **67–69**
  to uppercase names, 602–603
CONVERT.EXE utility, 33, 140
Convert function, 96
Convert Project to Model option, 437
Convert to Model option, 437
Convert tool, 33, 140
ConvertAndShow method, 141
ConvertIt demo, 100
ConvertService service, **880–882**, *881*, *883*
ConvUtils unit, 91, 96
CoolBar component, **230–231**
coordinates
  for controls, 159
  in dragging, 544
  for mouse-related keys, 262
  in Rave, 719
  in scrolling, **273–274**, *273*
Copy1Click method, 209
CopyFile method, 138–139
copying
  components, **24–25**
  files, 138–139
CORBA (Common Object Request Broker Architecture), 651
core library classes, **109**, **148**
  Classes unit, **147–148**
  collections, **131**
  containers, **131–135**
  events, **124–128**
  lists, **128–131**, *130*, **133–135**
  RTL package, **110–112**, *110*
  streaming, **135–148**, *141*
  TComponent, **117–124**, *120*
  TPersistent, **112–117**, *117*
core syntax of XML, **834–835**
Count command, 316
Count property, 128
count(*) wildcard, 561
CountBlanks property, 730
counters
  hit, **782–785**
  in Interbase 6, 565
CountSubstr function, 93
CountSubstrEx function, 93
CppDll example, 401–402, *402*
cracks in datasets, 697
Create constructors, 54
  for file streams, 138
  overloaded, 54–55
  in TMdClock, 351
  in TObject, 54
Create XML From Datapacket command, 856–857
CreateComObject function, 460, 465, 475
CreateComps example, 47–48, *47*
CreateDataSet method, 511
createElement method, 842
CreateFileMapping function, 413–414
CreateForm method, 77, 277
CreateGUID function, 87
CreateHandle method, 161
CreateInstance method
  in IClassFactory, 460
  in IObjectContext, 497
CreateMutex method, 308
CreateOleObject function, 475
CreateParams method, 161, 258
CreatePolygonalRgn method, 365
CreateTable method, 695
CreateWindowEx function, 258
CreateWindowHandle method, 161
CreateWnd method, 161, 327, 342–343
creation classes, 475
Creational Wizard, 452
CreatOrd program, 279
CSS (Cascading Style Sheets), **764–765**, 797, 859
CTS (Common Type System), 903–904, **909**
  events and delegates in, **909–910**
  objects and properties in, **909–910**
culture, 904
CUR files, 34
currency
  converting, **99–102**, *101*, **880–882**, *881*, *883*
  in Delphi for .NET Preview, 921
  floating-point numbers with, 87
Currency property, 658
CurrentYear function, 87

Cursor Service, 618
CursorLocation property, 629
cursors
  in ADO, **629**
    locations, **629–630**
    types, **630–631**
  unidirectional, **560–561**, **566–567**
CurValue property, 641–642
CustHint example, 183–184, *184*
CustLookup example, 519, *519*
custom attributes, **927–928**
custom classes for streaming, **142–145**
custom drawing technique, 185
custom method calls, **663–664**
custom variant types, 94–95, *94*
CustomConstraint property, 657
Customizable option, 244
CustomizeDlg component, 242, *244*
CustomNodes example, 197–198, *199*
CustQueP example, 774, 778–780, *780*

D

D7RegClean.exe utility, 33
data access components, **725–726**, *725*
data-aware components, **728–730**, *729*
data-aware controls, 162, **515**
  DBNavigator, **516**
  graphical, **520**
  grids, **516**
  list-based, **517–518**, *518*
  lookup, **519–520**, *519*
  mimicking, **545–547**, *545*
  replicable, **674**
  text-based, **516**
data binding interfaces, **846–850**, *847*
data connections in Rave, **721–722**, *722*
data links, **669–671**
  for ADO, **621**
  record viewer component, **678–683**, *682*
Data Lookup Security option, 726
data modules, **535**, 648
  for COM+, **497**
  for WebSnap, **798–799**
data packets, **651–652**, **667–668**
Data property
  for OLE objects, 481
  in TBucketList, 132
Data Text Editor, **728–729**, *729*
data types, COM, **478**
Data View Dictionary, 717, 723
DataBand component, 726–727
Database Connections option in Rave, 726
Database Desktop tool, 33
Database Explorer tool, 33
database independence, 615
databases
  ADO for, **508**
  BDE for, **507**, 615
  client/server programming for. *See* client/server programming
  ClientDataSet for. *See* ClientDataSet component
  data-aware controls for. *See* data-aware controls
  DataSet for. *See* datasets and DataSet component; fields
  dbExpress library for, **506–507**
  design issues
    entities and relations, **557–558**
    primary keys and OIDs, **558–560**
  error management for, **553–554**
  grids for, **540–544**, *542–543*
  multitier. *See* DataSnap
  platforms for, **567–569**
  reports for. *See* Rave
  sending data over socket connections, **744–747**, *747*
  standard controls, **544–545**
    data-aware-like, **545–547**, *545*
    for sending requests, **547–549**, *548*
  Web applications, **825–827**, *827*
    client side, **831–832**, *832*
    linking to details, **827–830**, *828*, *830*
  Web services for, **883–887**, *887*
  WebSnap application, **798**
    data editing, **801–803**, *802*
    data module for, **798–799**
    DataSetAdapter for, **799–801**, *800*
    master/detail in, **803–805**, *805*
DataChange method, 673
DataClone example, 633, *633*
DataCLX, 110–112
DataCycle component, 730
DataEvent method, 670
DataField property
  for data-aware controls, 515, 669, 672
  in Data Text Editor, 728–729
datagrams, 739
DataLinkDir function, 621

- DataMemo component, 729
- DataMirrorSection component, 732–733
- DataRelation class, 646
- DataSet component, *See* datasets and DataSet component
- DataSet property
  - for ClientDataSet connections, 509
  - in DataSetTableProducer, 762
- DataSetAdapter component, 794, **799–801**, *800*, **803–805**, *805*
- DataSetPageProducer component, 759, 761–762, 776, 789
- DataSetProvider component, 575, 653
- DataSetReader class, 646
- datasets and DataSet component, **520–524**
  - actions in, 224, **516**
  - class definitions for, **687–690**
  - closing, **694–695**
  - custom, **686–687**
  - in dbExpress, **572–574**, *573*
  - directories in, **705–709**, *709*
  - fields of. *See* fields
  - in IBX, **592–593**
  - initializing, **690–691**
  - navigating, **536–540**, **695–698**, *696*
    - bookmarks for, **537–538**
    - editing columns, **539–540**
    - totals of columns, **537**, *537*
  - of objects, **710–713**, *713*
  - opening, **690–694**
  - record buffers in, **698–703**
  - status of, **524–525**
  - testing, **703–704**, *704*
- DataSetTableProducer component, 759, 762–764, *763*, 776, 791
- DataSetToDOM method, 843–844
- DataSnap, **647**
  - ConnectionBroker in, **666**
  - custom method calls in, **663–664**
  - data packets in, **651–652**, **667–668**
  - example, **653**
    - client features in, **659–662**, *659*, *661*
    - first server, **653–655**, *654*
    - first thin client, **655–656**, *656*
    - server constraints in, **657–658**
  - IAppServer interface for, **649–650**
  - levels in, **647–649**
  - master/detail relations in, **664–665**, *665*
  - object pooling in, **667**
  - parametric queries in, **663**, *663*
  - protocols for, **650–651**
  - provider options in, **666–667**
  - SimpleObjectBroker in, **667**
  - over SOAP, **889–892**
  - support components for, **652–653**
  - technical foundation of, **649**
- DataSource component, 575, 669
- DataSource property
  - for data-aware controls, 515, 669
  - in SQLDataSet, 581
  - in TDataSet, 552
  - in TMdDbProgress, 672
- DataTable class, 646
- DataView property
  - in Data Text Editor, 729
  - in DataBand, 726
- DateCopy example, 57–58
- DateList example, 134–135
- DateProp example, **51**, *51*
- dates
  - conversions with, 87
  - native formats for, 709
  - unit for, 91
- DateTimeToNative function, 709
- DateTimeToSQLTimeStamp function, 574
- DateUtils unit, **91**
- DAX (Delphi ActiveX) framework, 112
- DaysBetween function, 91
- DbAware example, 517–518, *518*
- DBCheckBox component, 515, 517
- DBComboBox component, 517
- DBCtrlGrid component, 674
- DBEdit component, 515–516
- DBError example, 553–554
- dbExpress Connection Editor, 571, *571*
- dbExpress Draft Specification, 567
- dbExpress library, **506–507**
  - client/server programming with, **566**
    - components, **569–574**, *571*, *573*
    - metadata in, **578–579**, *579*
    - parametric queries in, **579–581**, *580–581*
    - platforms and databases, **567–569**
    - printing in, **581–584**
    - single and multiple components with, **575–578**, *577*
    - unidirectional cursors, **566–567**
  - components
    - dataset, **572–574**, *573*
    - SQLConnection, **569–574**, *571*
    - SQLMonitor, **574**
  - drivers for, **567–569**

dbGo package, 508, 616, **618–620**, *619–620*
DBGrid control, **515–516**, **544**
    customizing, **683–686**, *683*
    dragging with, **544**
    with multiple selections, **542–543**, *543*
    painting, **540–542**, *542*
DBI files, 38
DBImage component, 520
DBListBox component, 517
DBLookupComboBox component, 518–519
DBLookupListBox component, 518
DBMemo component, 516
DBNavigator component, **515–516**
DbProgr example, 673, *673*
DBRadioGroup component, 517–518, *518*
DBText component, 516
DbTrack example, 677, *677*
DBX. *See* dbExpress library
DbxExplorer example, 579
DbxMulti example, 568, 575–578
DbxSingle example, 575–578, *577*
DCC.exe tool, 33
dccil compiler, 900
DCI files, 37
DCOM (Distributed COM), 650
DCOMConnection component, 652, 655
DCOMConnection1 class, 655
DCP files, 35, 347
DCR (Delphi Component Resource) project type, 348
DCT files, 38
DCU (Delphi Compiled Unit) files, 30, 35, 338–339, 416
.dcua files, 906–907
.dcuil files, 906–907
DDE (Dynamic Data Exchange), 456
DDP (Delphi Diagram Portfolio) files, 35
debugger, **42**
Debugger Options dialog box, 74
debugging, 33
    and exceptions, **74**
    SOAP headers, **887–888**, *888*
    in WebBroker technology, **772–774**, *774*
Decision Cube components, 112
declarations, external, 398, 406
_declspec declaration, 401
DecodeDateFully function, 87
DecompressStream function, 146
Decorator pattern, 449
DefAttributes property, 164
default exception handlers, 74
default keyword for styles, 343
default values for methods, 46
DefaultColumnWidth property, 680
DefaultDrawColumnCell method, 540
DefaultDrawing property, 540, 682
DefaultExpression property, 657
DefaultRowHeight property, 683, 685
DefaultStyle property, 219
DefaultTextLineBreakStyle variable, 85
DefaultTimeout property, 805
Define command in Fields editor, 526
DefineProperties method, 113, 187, 683
DefinePropertyPage method, 491
DefinePropertyPages method, 491
defining
    actions, **385–388**
    events and event handlers, **364–365**
    symbols, 156
definitions
    external, 400
    links to, 13
delayed signing, 917
delegates in CTS, **909–910**
delegation in event handling, 124
Delete method, 128
delete operations, trigger firing from, 565
Delete1Click method, 209
Delphi ActiveX (DAX) framework, 112
Delphi Compiled Unit (DCU) files, 30, 35, 338–339, 416
Delphi compiler tool, 33
Delphi Component Palette, 287
Delphi Component Resource (DCR) project type, 348
Delphi Diagram Portfolio (DDP) files, 35
Delphi for .NET Preview, **900–902**, *903*, **919**
    ASP.NET with, **940–942**, *942–943*
    class helpers in, **929–931**
    custom attributes in, **927–928**
    deprecated features for, **920–922**
    extended identifiers in, **924**
    final and sealed keywords in, **924–925**
    Microsoft libraries for, **934–940**, *940*
    multicast events in, **926–927**
    nested types in, **926**
    run-time library for, **930–931**
    static members in, **925–926**
    unit namespaces in, **922–924**
    VCL for, **931–933**, *933*
    visibility and access specifiers in, **925**
Delphi internal messages, 374

Delphi language, **43–44**
  class references in, **76–78**, *78*
  classes and objects in, **44–48**
  for components, 339
  constructors in, **54–55**
  dynamic objects in, **47–48**, *47*
  encapsulation in, **48**
    access specifiers for, **48–50**, **60–62**
    and forms, **52–54**, *53*
    with properties, **50–52**, *51*
  exceptions in, **71–75**, *76*
  inheritance in, **59–63**, *60*
  interfaces in, **69–71**
  late binding in, **63–67**, *64*
  type-safe down-casting in, **67–69**
Delphi Power Book, 954
Delphi project (DPR) files, 39
DELPHI32.DCT file, 26
DelphiMM unit, **95–96**
Delphree site, 952
delta caches, 652
Delta memory area, 514
delta packets, 651
Delta property, 585–586, *586*
deltas
  in aggregates, 550
  in batch updates, 638
  in ClientDataSet, **585–586**, *586*, 652
DEM files, 38
demoscript pages, 792–793, *792*
dependencies, 339
deployment
  in library selection, 155
  in .NET architecture, **916–917**, *917*
deployment diagrams, 434
deprecated features, **920–922**
descendant classes, 63
DESCRIPTION directive, 346
Description property, 353
design critics, 452
design-only component packages, 338
design patterns, **447–450**
design-time packages, 338
design-time properties, 52
Designer page, 19, 280
Designer toolbar in Rave, 718
Desktop (DSK) files, 36, 39
desktop settings, saving, **5–6**
Dessena, Nando, 600
Destination property, 249
DestParan property, 733
Destroy method, 55, 59, 912–914, *914*
DestroyComponents method, 118
destroying objects, **58–59**
destructors, **55**, 59
DestructorTest class, 912–914, *914*
DetailKey property, 731
details, linking, in Web databases, **827–830**, *828*, *830*
Details view in Object Repository, 40
~DF files, 35
DFM files, 19, 33, 35, 38, **152–154**
DFN files, 35
Diagram view, **16–18**, *17–18*
diagrams
  for classes, **431–432**, *432–433*
  common elements in, **435–436**, *436*
  non-UML, **435**
  for sequences, **433–434**, *433*
  for use cases, **434**
dialog actions, 224
dialog boxes, **280–281**
  creating, **282–287**, *284–285*
  modeless, **285–287**, *285*
  nonvisual, **378–381**, *381*
  predefined, **287–289**, *288*
dialog menu items, 173
diamond symbol in Band Style Editor, 728
Difference view, **442–443**, *443*
digital clock, 349–352
DirDemo example, 709, *709*
Direct Data View option, 726
Direct Driver View option, 726
direct form input, **259**
  keyboard, **259–261**, *260*
  mouse, **261–265**
direct memory access functions, 921
Direction property, 489
directives, 44, 347
directories
  for components, 347
  in datasets, **705–709**, *709*
Directories/Conditionals page, 477
DisableCommit method, 497
DisableControls method, 538–539
DisabledImages property, 213
DisableIfNoHandler property, 223, 227
disabling aggregates, 551
disconnected recordsets, **642–643**

dispatching calls in Automation, **468–470**, *469*
dispid keyword, 470
dispinterface keyword, 469–470, 472, 488
Display method, 422
DisplayFormat property, 658–659, 733
    in DataSetTableProducer, 763
    in TFloatField, 527, 532
DisplayLabel property, 527–528, 658
DisplayType property, 801
DisplayValues property, 658
DisplayWidth property, 528, 658
Dispose method, 912
Distributed COM (DCOM), 650
distribution of updated packages, 416
DivideTwicePlusOne example, 73–75
DivMod function, 89
DlgApply example, 285–287, *285*
DllCanUnloadNow function, 461
DLLGetClassObject method, 460
DllMem example, 412–413
DllRegisterServer function, 461
DLLs (dynamic link libraries), 35, **397**
    for ActiveX controls, 484
    C++, **401–402**
    calling, **406**, *407*, **408–410**
    creating, **402–406**
    dynamic linking in, **397–398**
    exporting strings from, **404–406**
    forms in, **410–411**
    with ISAPI, **769–770**
    in memory, **411–414**, *415*
    overloaded functions in, **404**
    and packages, 30, 338, 345, **415–416**
    purpose of, **398–399**
    rules for, **399–400**
    using, **400–402**, *402*
    wizards for, 41
DllUnregisterServer function, 461
DMT files, 37
doAutoIndent option, 842
DoChange method, 126–127
Dock method, 235
DockClientCount property, 235
DockClients property, 235
docking support, **234–235**
    in control bars, **235–239**, *236*
    messages for, 376
    for page controls, **239–241**, *241*
DockPage example, 239–241, *241*
DockSite property, 234
DoConvert method, 98–99
DoCreate method, 330–331
document management in XML, **837–838**, *840*, *843*, *846*
Document Object Model (DOM)
    programming with, **838–846**, *840*, *843*, *846*
    XSL transformations with, **868–869**
document type definitions (DTDs), 849
documentation in ModelMaker, **444–445**, *444*
DoDestroy method, 331
Does Not Support Transaction option, 495
DOF files, 28, 35
DOM (Document Object Model)
    programming with, **838–846**, *840*, *843*, *846*
    XSL transformations with, **868–869**
domain names, **739**
DomCreate example, 839, 842–846, *843*, *846*
DOMDocument property, 838
DOMPersist property, 836
DOMVendor property, 836
dot-notation for methods, 46
Double type, 478
DoVerb method, 481
down-casting, type-safe, **67–69**
~DP files, 36
DPK files, 35, 345
DPKL files, 35
DPKW files, 35
DPR (Delphi project) files, 36, 39
dragging, 98
    with DBGrid control, **544**
    with mouse, **262–265**, *265*
    in Object TreeView, **23**
    source code files, 27
DragKind property, 234–235
DragMode property, 98
    in TControl, 234–235
    in TreeView, 195
DragToGrid example, 544
DragTree example, 194–197, *194*
DrawCell method, 680–682
DrawColumnCell, 685–686
DrawData program, 540–542, *542*
DrawFocusRect method, 264
drawing
    with mouse, **262–265**, *265*
    Rave components for, **725**

DrawPoint method, 263
DrawText function, 209, 682
DriverName property, 570
drivers for dbExpress, **567–569**
drivers.ini file, 569–570
Drivers Settings window, 571, *571*
DRO files, 37
drop-down fonts, **21**
drop procedure statement, 565
drop trigger statement, 566
DropdownMenu property, 213, 215
DropDownRows property, 516
DropDownWidth property, 519
dsBrowse value, 524
dsCalcFields value, 524
dsCurValue value, 524
dsEdit value, 524
dsFilter value, 524
dsInactive value, 524
dsInsert value, 524
DSIntf.pas unit, 510
DSK (Desktop) files, 36, 39
DSM files, 36
dsNewValue value, 524
dsOldValue value, 524
DST files, 6, 38
DTDs (document type definitions), 849
dual interfaces, 470
dual libraries support, **151–155**, *152*
DUnit architecture, 952
DynaCall example, 408–410
DynaForm example, 252–253, *253*
dynamic aggregation of interfaces, 921
dynamic binding. *See* late binding
dynamic components, **47–48**, *47*
dynamic cursors, **631**
Dynamic Data Exchange (DDE), 456
dynamic database reporting, **776–777**, *777*
dynamic link libraries. *See* DLLs (dynamic link libraries)
dynamic linking, **397–398**
dynamic methods vs. virtual, **65–66**
dynamic pages, **764–765**
dynamic properties, **622**
dynamic Web pages, **768**
    and CGI, **768–769**, *769*
    DLLs, **769–770**
DynaPackForm example, 419–420

E

e-mail, **748–750**, *749*
E object, 75
early binding, 63
ebXML (Electronic Business using Extensible Markup Language), 894
EchoActionFieldValue property, 797
EchoMode property, 163
ECMAScript, 791
EDBClient class, 553
Edit command, 520
Edit component, **163**
Edit method
    in IComponentEditor, 393
    in TSoundProperty, 389–390
Edit Tab Order dialog box, 176, *176*
Edit To-Do Item window, 8, *8*
edit verbs, 480
EditFirstNameExit method, 178
EditFormat property, 658
editing
    actions for, 224
    columns, **539–540**
    connection properties, 571
    HTML, 788
    in-place, 480
    records, 520
    textual form descriptions, 38
    visual, 484
    WebSnap data, **801–803**, *802*
editions, **3–4**
EditLabel property, 163
EditMask property, 163–164, 658
Editor Properties dialog box, 10–11, *10*
editors, **10–11**, *10*
    ActionManager, 242–245, *243–245*
    for columns, 533
    for commands, 573, *573*
    for components, **392–395**, *394*
    for connection strings, 618–619, *619–620*
    Delphi
        browsing in, **12–13**
        class completion in, **13–14**
        Code Explorer window in, **11–12**, *12*
        code insight technology in, **14**
        loadable views in, **16–18**, *17–18*
        shortcut keys in, **16**
    for fields, 525–527, *526*, 550

  - for images, 33, 348
  - for input masks, 163–164
  - Method Implementation Code Editor, **441–442**, *442*
  - for packages, 345, *345*, 347–348, 417, *417*
  - for properties, **388–392**, *390–391*, 533
  - type-library, **470–472**, *471*
  - for unit code, **439–441**

EDivByZero exception, 73
Electronic Business using Extensible Markup Language (ebXML), 894
ellipsis (...) on menus, 173
embedded units, **568–569**
embedding, 480–481
EMdDataSetError class, 687–688
Empty Project template, 40–41
empty values with field events, 535
Enable property, 863
EnableCommit method, 497
EnableControls method, 538–539
Enabled property
  - for actions, *222*, *227*
  - in TControl, 159
  - in Timer, 350

encapsulation, **48**
  - access specifiers for, **48–50**, **60–62**
  - and forms, **52–54**, *53*
  - with properties, **50–52**, *51*

end-of-file cracks, 697
endDocument event, 850, 852
endElement event, 850, 872
EndThread function, 84
EndUser object, 793, 795
EndUserSession adapter, 795
EndUserSessionAdapter component, 807–808
EnlargeFont1Click method, 816
EnsureRange function, 89
Enterprise Studio edition, 4
entities in database design, **557–558**
enumerated properties, **358–360**
EnumModules function, 84, 425–426
EnumWindows functions, 308
EnumWndProc method, 308–309
Environment Options dialog box, **6–7**
  - for AutoSave feature, 11
  - for Code Explorer, 11–12
  - for compiler, 30
  - for DFM file saving, 19
  - for HTML editing, 788
  - for Project Explorer, 32
  - for secondary forms, 280
  - for type libraries, 471

environment variables in CGI programs, 769
Environment Variables page, 7
Eof tests, 536
EofCracks, 697
EqualsValue constant, 89
equi-joins, 636
error method, 851
ErrorLog example, 75–76, *76*
errors. *See also* exceptions
  - in ADO, 641
  - database, **553–554**
  - logging, **75–76**, *76*

Essential Delphi, **954**
Essential Pascal, **953–954**
etm60.exe tool, 33
EuroConv example, 100–102, *101*
EuroRound function, 102
euros, converting, **100–102**, *101*
event-driven programming, **302–303**
Event Editor in Rave, 717, 731–732
Event Types view, **443**
EventHandler method, 162
events, **124**
  - for ActiveX controls, 484
  - for applications, 296
  - for Arrow component, **364–366**
  - for COM+, **497–500**, *499*
  - in CTS, **909–910**
  - in DataSnap example, **658–659**
  - defining, **364–365**
  - in Delphi for .NET Preview, **926–927**
  - in Interbase 6, **565–566**
  - method pointers for, **125–126**
  - null values with, **535–536**, *536*
  - as properties, **126–128**
  - in Qt, 162

EvtSubscriber library, 498
Excel
  - with Automation, 479
  - Jet engine for, **625–626**, *626*

Excel IISAM, 626
except blocks, 71–73
Exception class, 75
Exception1 example, 73–75
exceptions, **71**
  - in calculated fields, 532
  - classes for, **73–75**

  - for components, 339
  - and debugging, **74**
  - error logging, **75–76**, *76*
  - program flow and finally block in, **72–73**
- Exclude method, 927
- EXE files, 36
- ExecSQL method, 572
- executable files, 30, **82–83**
- executable viewer tool, 33
- Execute method
  - in TFindWebThread, 753
  - in TMdListBoxDialog, 380
  - in TMdListCompEditor, 395
  - in TPrimeAdder, 305–306
- ExecuteTarget method, 385–387
- ExecuteVerb method
  - in IComponentEditor, 393
  - in TMdListCompEditor, 394
- exitItemClick method, 937, 939–940
- ExitProc function, 921
- ExpandAll1lick method, 816
- ExpandFileName function, 87
- expanding component references in-place, 20
- explicit transactions, 589
- Explorer page, 32
- exploring projects, **32**, *32*
- exporting
  - Jet engine for, **628–629**
  - strings from DLLs, **404–406**
- exports clause
  - in Apache, 782
  - for DLLs, 399, 404
- expression evaluation, tooltip, 16
- ExtCtrls unit, 163
- Extended Database Form Wizard, 947
- extended identifiers, **924**
- extended ListBox selections, 167
- Extended Properties property, 624, 626, 629
- ExtendedSelect property, 167
- extending libraries, **337–340**
- Extensible Stylesheet Language (XSLT), **864**
  - examples, **865–866**
  - with WebSnap, **866–868**, *866*
  - XPath in, **864–865**
- external components
  - in compound components, **352–354**, *353*
  - interfaces for, **354–356**
- external declarations, 398, 406
- external definitions for DLLs, 400
- external keys, **559**
- external symbols, 15
- External Translation Manager tool, 33
- $EXTERNALSYM directive, 400
- ExtraCode property, 197
- ExtractStrings method, 147
- Extras key for hot-track activation, 24

F

- fake implementation code, 498
- fArrowDblClick method pointer, 364–365
- fatalError method, 851
- FCL (Framework Class Library), **904**
- FetchBlobs method, 584, 666
- FetchDetails method, 584, 666
- FetchOnDemand property, 666
- FetchParams method, 663
- field-oriented data-aware controls
  - read-only progress bar, **671–673**, *673*
  - read-write TrackBar, **674–677**, *677*
- Field property, 671
- FieldAcc example, 527–528, *528*
- FieldAddress method, 115
- FieldByName method, 525, 527
- FieldDefs property, 511, 690
- FieldLookup example, 534, *534*
- FieldName property
  - in TField, 527
  - in TFieldDataLink, 671
- fields
  - for adapters, **794**
  - in databases, **525–527**, *526*, **698–702**
    - calculated, **530–533**, *531*, *533*
    - class hierarchy for, **529–530**
    - constraints on, **657–658**
    - events for, **658–659**
    - lookup, **533–534**, *534*
    - null values with events for, **535–536**, *536*
    - objects for, **527–528**, *528*
    - properties for, **658**
  - in Delphi for .NET Preview, 926
  - hiding, **122–123**
  - names for, 360
  - removing, **121–122**
  - virtual, 50
- Fields editor, 525–527, *526*, 550
- Fields property, 525, 527

- FieldsToXml function, 886
- FieldValues property, 525, 527
- file management, **102–104**, *103–104*
- File Transfer Protocol (FTP)
  - API for, 754–755
  - ports for, 739
- FileCreate function, 87
- FileName property, 244
- files
  - actions for, 224
  - copying, 138–139
  - locating, **798**
  - produced by system, **34–38**
  - source code, **38–39**
  - streaming, **138–139**
- FilesList example, 102–104, *103–104*
- Filled property, 359, 366–367, 489
- FillFormList method, 299–300
- FillFormsList method, 300
- Fills toolbar in Rave, 718
- Filter property, 513
- FilterGroup property, 638, 641
- filtering for ClientDataSet, **513**
- final keyword, **924–925**
- Finalize method, **911–912**
- finally blocks, **71–73**
- Find and Replace dialog boxes, 288
- find_business API, 897
- Find method, 132
- FindClose function, 102
- FindComponent method, 121, 286–287, 765
- FindFirst function, 102
- FindGlobalComponent function, 121
- FindNearest method, 777
- FindNext function, 102
- FindNextPage function, 205–206
- FindWindow function, 307
- FirstDll example, 405–406
- FishClient example, 893, *894*
- Flat property, 213
- floating-point numbers
  - comparisons with, 89
  - conversions with, 87
- FloatingDockSiteClass property, 234
- FloatToCurr function, 87
- FloatToDateTime function, 87
- fly-by-hints, **182–184**, *184*, 218, *218*
- fm flags, 138
- FocusControl property, 177
- Font dialog box, 288, *288*
- Font property, 21, 159
- FontDialog component, 288
- FontMaster component in Rave, **724**
- FontMirror property, 724
- FontNamePropertyDisplayFontNames variable, 21
- fonts
  - in Automation, **478**
  - for controls, **159–160**
  - for dialog boxes, 288, *288*
  - listing, 299
  - for menu items, 216
  - in Object Inspector, **21**
  - size of, 276–277
  - with toolbars, 215
- Fonts combo box, **341–344**
  - package for, **344–347**, *345–346*
  - using, **347–348**
- Fonts toolbar in Rave, 718
- ForEach method, 133
- ForEachFactory method, 462
- ForEachModule function, 425–426
- Form component, 119
- Form Designer, **18–19**
- Format function, 85, 582
- FormatDateTime function, 60
- formats for Rave, **720–721**, *720*
- FormatXMLData method, 842, 857
- FormCount property, 821
- FormCreate method, 127
  - for pages, 204
  - in TDateForm, 127
  - in TForm1, 196–197
  - in TFormUseIntf, 423
- FormDestroy method, 131
- FormDLL example, 411
- FormIntf demo, 330–331
- FormMouseDown method, 263–264
- FormMouseMove method, 263–264
- FormMouseUp method, 263, 265
- FormProp example, 53, *53*
- forms, **251**
  - About boxes and splash screens for, **289–292**, *290*
  - activating, **298**, *298*
  - border icons for, **256–257**, *257*
  - borders for, **254–256**, *254*
  - client area in, **269**

- closing, **279–280**
- constraints in, **270**
- creating, **277–279**, *278*
- dialog boxes
  - creating, **282–287**, *284–285*
  - predefined, **287–289**, *288*
- direct input
  - keyboard, **259–261**, *260*
  - mouse, **261–265**
- displaying, **267–268**, *267*
- in DLLs, **410–411**
- and encapsulation, **52–54**, *53*
- fields in
  - hiding, **122–123**
  - removing, **121–122**
- inheritance in
  - from base forms, **319–321**, *319–320*
  - polymorphic, **321–324**, *322*
- multiple-page, **202**
  - image viewer in, **207–210**, *207*
  - PageControl and TabSheet for, **202–207**, *204–205*
  - for user interface wizard, **210–213**, *211*
- in packages, **417–419**, *417*
- painting in, **265–267**
- plain, **252–253**, *253*
- position of, **268–269**
- properties for, **53–54**, *53*
- scaling, **274–277**
- scrolling, **270–271**
  - automatic, **272–273**
  - coordinates in, **273–274**, *273*
  - example, **271–272**, *272*
- secondary, **280–281**
- size of, **269**
- snapping in, **269**
- splitting, **180–181**, *180*
- style of, **254**
- TForm class, **251–259**, *253–254*, *257–258*
- tracking, **299–302**, *300*
- WebBroker technology for, **777–781**, *780*
- windows for, **257–259**, *258*

Forms page, 28, 277, *278*, 280
FormStyle property, 254, 257, 311
FormToText program, 140–141, *141*
forward-only cursors, **630–631**
Frac function, 921
FramePag example, 327–328, *327*
frames, **324–326**, *325*
- in Component Palette, **26**, *26*
- for compound components, **357–358**
- in MDI applications, **311–315**, *313*
- with pages, **326–328**, *327*
- without pages, **328–330**, *329*

Frames1 example, 26, *26*
Frames2 example, 324–326, *325*
FrameTab example, 328–330, *329*
Framework Assembly Registration Utility (regasm), 501
Framework Class Library (FCL), **904**
Free method, **55–56**, 58–59, 912
free query forms, **613**, *613*
Free threading model, 463
FreeAndNil method, 55, 59
FreeInstance class, 335
FreeLibrary function, 408–409
FreeMem method, 84, 921
FreeNotification method, 302, 354, 356
FreeRecordBuffer method, 699
FROM clause in SELECT, 628
FromStart property, 826
fs constants, 254, 311
FTimer class, 351
FTP (File Transfer Protocol)
- API for, 754–755
- ports for, 739

FullExpand method, 194
functions
- addresses of, 63
- names for, 360
- overloaded, 46, **404**

G

GAC (Global Assembly Cache), 917, *917*
GACUTIL utility, 917
garbage collection, **911–916**
Garbage Test example, 915–916
gen_id function, 566
GeneratorField property, 593, 601
generators, **565–566**, **600–602**, *601*
generic projects, 922
generic unit references, 923
generic units, 922
Get methods
- in TAXForm1, 491
- in WebBroker, 778

Get_Value method, 473
GetAttributes function, 388–389
GetBookmarkData method, 696
GetBookmarkFlag method, 696
GetBufStart method, 194
GetCanModify method, 706
GetCds method, 892
GetClass function, 419
GetCollString method, 384
GetColor function, 410–411
GetControl function, 386–387
GetData method, 547, 702
GetDataAsXml method, 857
GetDataField method, 672
GetDataSource method, 672
GetDriverFunc function, 570
GetDriverVersion method, 568
GetEmployeeData method, 885–886
GetEmployeeNames method, 884–885
GetEnumName function, 585
GetEnvironmentVariable function, 87
GetExtraCode function, 197
GetField method
    in TMdDataSetOne, 703
    in TMdDbProgress, 672
GetFieldData method
    in TMdDataSetOne, 702–703
    in TMdDirDataset, 708
    in TObjDataSet, 711–712
GetFileVersion function, 88
GetHeapStatus function, 84
GetIDsOfNames method, 468
GetInterfaceExternalName, 882
GetKeyState function, 218
GetLines method, 379–380
GetLocaleFormatSettings function, 88
GetLongHint method, 218
GetMem method, 84, 921
GetMemo function, 335
GetMemoryManager function, 84, 334
GetMethExternalName, 882
GetModuleFilename, 309
GetNamePath method, 113
GetNewId function, 601
GetNextPacket method, 584
GetObjectContext method, 497
GetOleFont method, 478
GetOleStrings function, 478
GetOptionalParameter method, 668
GetOwner method
    in TMdMyCollection, 382–383
    in TPersistent, 113
GetPackageDescription function, 424
GetPackageInfo function, 424, 426–427
GetPackageInfoTable function, 84, 424
GetProcAddress function, 408, 412
GetPropInfo function, 116
GetPropValue function, 116
GetRecNo method, 698
GetRecord method, 699–700
GetRecordCount method, 698, 891
GetRecordSize method, 699
GetSelItem function, 380
GetShareData method, 414
GetShortHint method, 218
GetStrProp function, 116
GetSubDirs function, 103
GetSystemMetrics function, 259
GetText method, 59–60
GetTextHeight method, 187
GetTickCount function, 476
GetValue method, 390
GetValues method, 389
GetVerb method
    in IComponentEditor, 393
    in TMdListCompEditor, 394
GetVerbCount method, 393
GetWindowLong function, 298
GetYear method, 51
GExperts tool, **952**
Global Assembly Cache (GAC), 917, *917*
global data
    for DLLs, 400
    in Project Explorer, 32
    in System unit, 84
global deployment, 916
global handlers, 72
Global Page Catalog node, 717, 723
GlobalCount variable, 821
GlobalEnter method, 178
Globally Unique Identifiers (GUIDs)
    conversions with, 87
    in DCOMConnection, 655
    in Delphi for .NET Preview, 928
    for interfaces, 69
    in IUnknown, **458–460**, *459*

Google website, 751–752
GotoPage method, 485
GrabHtml method
    in TFindWebThread, 752–753
    in WebSearch program, 785
graphic toolbars, 213
graphical components in Rave, **724**
graphical controls, 158. *See also* images
    compound components. *See* Arrow component
    data-aware, **520**
    hit counters, 783–784
GreaterThanValue constant, 89
Green Pages in UDDI, 895
Grep.exe tool, 33
GridDemo example, 686
grids
    data-aware controls for, **516**
    for databases. *See* DBGrid control
GroupBox component, 119, **166**
Grouped property, 230
GroupIndex property
    for actions, 230
    for menu items, 481
    for radio items, 173
groups
    in ClientDataSet, **549**, *550*
    project, 26–27
GUIDs (Globally Unique Identifiers)
    conversions with, 87
    in DCOMConnection, 655
    in Delphi for .NET Preview, 928
    for interfaces, 69
    in IUnknown, **458–460**, *459*
GUIDToString function, 459

H

H files, 401–402
hacks, 62
Handle property
    in Application, 297
    in TWinControl, 161
HandleMessages function, 303–304
HandleNeeded method, 161
HandleReconcileError function, 588
HandlesPackages lists, 422
HandlesTarget method, 385
hashed associative lists, **132–133**
header files, 401–402
headers in CGI programs, 769
Height property
    for components, 18
    for forms, 268–269
    in TControl, 159
    in TMdArrow, 359
HelloWorld example, 901–902, *903*
Help actions, 224
Help files, 33, 340
HelpContext property, 222, 256
Hide method, 159
HideComp example, 122–123
HideOptions property, 808
HideUnused property, 246
hiding
    components, 159
    form fields, **122–123**
    forms, 279
high-level protocols in socket programming, **740**
high-low technique, 600
hint directives, 44
Hint property
    for actions, 222
    in Application, 217
    for toolbars, 236
HintColor property, 182
HintHidePause property, 182
HintPause property, 182
hints in ToolBar, **182–184**, *184*, 218, *218*, 236
HintShortPause property, 182
hit counters, **782–785**
HitsGetValue method, 806
HookEvents method, 162
horizontal splitting of forms, **181**
HorzScrollBar property, 270, 273
hosts file in IP addresses, 739
hosts in socket programming, 741
HOSTS.SAM file, 739
hot-track activation, 24
HotImages property, 213
Hower, Chad Z., 809
HTM files, 36
HTML (Hypertext Markup Language), **758–759**
    for data, **761–762**, *762*
    for dynamic pages, **764–765**
    for pages, **759–761**, *760*
    producer components for, **759**

  style sheets for, **764–765**
  for tables, **762–764**, *763*
  templates for, 788
HTML editor, 788
HTML Import, 629
HTML tab, 862
HTMLDoc property, 788
HtmlDom component, 868
HtmlEdit example, 165, *165*
HTMLFile property, 788
HtmlProd example, 759–764, *760*, *762*
HtmlStringToCds method, 785
HtmlToList method, 752–754
HTTP (Hypertext Transfer Protocol), 651, **750–751**
  API for, 754
  for browsers, **756–757**, *757*
  grabbing content in, **751–754**, *755*
  ports for, 739
  server, **757–758**, *758*
  WinInet API for, **754–756**
HttpDecode function, 743
HttpEncode function, 743
HTTPRIO component, 879
HttpServ example, 757–758, *758*
HTTPSoapDispatcher component, 879
HTTPSoapPascalInvoker component, 879
httpsrvr.dll file, 651
HTTPWebNode property, 893
HttpWork method, 754
Hypertext Markup Language. *See* HTML (Hypertext Markup Language)
Hypertext Transfer Protocol. *See* HTTP (Hypertext Transfer Protocol)

I

IAppServer interface, **649–650**, 667, 890–891
IAppServerOne interface, 654
IAppServerSOAP interface, 891
IB provider, 618
IB_WISQL tool, 564
IBClasses dataset, 610
IBClassReg dataset, 609–610
IBConsole application, **562–564**, *563–564*
IBDatabase component, 592
IBDatabaseInfo component, 592
IBDataSet component, 592
IBEvents component, 592
IBM DB2 drivers, 567
IBPeopleReg dataset, 610
IBQuery component, 592
IBSQL component, 592
IBSQLMonitor component, 592, 598
IBStoredProc component, 592
IBTable component, 592
IBTransaction component, 592
IBUpdateSQL component, 592
IBX. *See* InterBase Express (IBX)
IbxEmp example, 593–595
IbxMon example, 598, *599*
IbxUpdSql example, 595–597, *598*
ICO files, 34
IComponentEditor interface, 393
icons
  for applications, 297
  for borders, **256–257**, *257*
IdAntiFreeze component, 738
IDAPI (Independent Database Application Programming Interface), 615
IDE (Integrated Development Environment), **3–4**, *5*
  compiler messages, **9–10**
  Component Palette. *See* Component Palette
  debugger, **42**
  desktop settings in, **5–6**
  editor. *See* editors
  environment settings in, **6–7**
  Form Designer, **18–19**
  libraries in, **5**
  menus, **7**
  Object Inspector, **19–23**
  project management in, **26–32**, *27*, *32*
  to-do lists, **7–9**, *8*
  tools, **33–34**
identifiers, **924**
IDispatch interface, 84, 458, 467–469, 851
IDisposable interface, 912, 915
idle state, 302
IdMessage class, 748
IDockManager interface, 235
IDOMAttr interface, 838
IDOMDocument interface, 838
IDOMElement interface, 838
IDOMNode interface, 838
IDOMNodeList interface, 838
IDOMParseError interface, 836
IDOMText interface, 838
IdPop3 component, 748

IDs
  GUIDs. *See* GUIDs (Globally Unique Identifiers)
  in Interbase, **600–602**, *601*
IdSMTP component, 748
IdTCPClient component, 741
IdTCPServer component, 741
idThreadMgrDefault component, 738
idThreadMgrPool component, 738
IeFirst example, 860–864, *861*, *863*
$IF directive, 44
IFirstServer interface, 471–473
IFontDisp interface, 478
IFormOperations interface, 333
IfSender example, 106–107, *106*
IfThen function, 88–89, 92
ignorableWarning method, 851
IIDs (interface IDs), 459
IInterface interface, 69, 84, 457
IInvokable interface, 84, 878
IISAM (Installable Indexed Sequential Access Method) drivers, 624
IL Disassembler (ildasm.exe), 908
ildasm.exe (IL Disassembler), 908
Image Editor tool, 33, 348
image viewer in multiple-page forms, **207–210**, *207*
ImagEdit.exe tool, 33
ImageIndex property
  for actions, 222
  in ListView, 189
  for menu items, 174
ImageList class, 189, 203, 213
images
  for dialog boxes, 288
  in ListView controls, **188–193**, *191*
  on menus, 185
Images property
  for menu items, 174
  for toolbars, 213
IMdInform interface, 498
IMdWArrowX interface, 488
implementation diagrams, 434
Implementation File Update Wizard, 495
implementation section
  for DLLs, 400
  uses statement in, 256
implicit transactions, 589
Import Source File dialog box, 438
Import Type Library dialog box, 477
ImportedConstraint property, 657
importing
  Jet engine for, **628–629**
  in ModelMaker, **437–438**
IN clause in SELECT, 628
in-place activation, 484
in-place editing, 480
in-place expansion of component references, 20
in-process servers, 457
Include method, 927
IncludePathURL property, 860
IncludeTrailingPathDelimiter function, 87
Increase method, 49, 495
Increment property, 270
incremental searches, shortcut keys for, 16
indenting, shortcut keys for, 16
Independent Database Application Programming Interface (IDAPI), 615
IndexDefs property, 513
indexes
  in ADO, **632**
  in ClientDataSet, **512–513**
IndexFieldNames property, 512, 632
Indexing Service, 618
IndexOf method, 128
Indy (Internet Direct) components, **738**, **741–744**, *744*
  HTML. *See* HTML (Hypertext Markup Language)
  HTTP. *See* HTTP (Hypertext Transfer Protocol)
  for mail, **748–750**, *749*
Indy (Internet Direct) project, 951
IndyDbSock example, 747, *747*
IndySock1 example, 741–744, *744*
INetXCustom, 859
InetXPageProducer, 858–859, 861–863, *861*
InetXPageProducer editor, 863
infinite constants, 88
Infinity constant, 88
InflateRect function, 682
InFocus example, 176–178, *177*
information hiding, 48
inheritance
  from existing types, **59–63**, *60*
  form, **318–319**
    from base forms, **319–321**, *319–320*
    polymorphic, **321–324**, *322*
  in ModelMaker, **437–438**
  restricting, 447
  type compatibility in, **62–63**
inherited keyword, 65, 272, 320–321
InheritsFrom method, 67, 105

INI files, **331–332**
initialization
  of COM objects, **464**
  constructors for, 54
  of datasets, **690–691**
initialization section, 473–474
Initialize method
  for automation servers, 474
  in TNumber, 464
InitializeControls method, 936–937
InitializeMenu method, 936, 939–940
InitWidget method, 162
inline keyword, 325
inner transactions, 634–635
input, forms
  keyboard, **259–261**, *260*
  mouse, **261–265**
input focus
  handling, **176–178**, *176–177*
  messages for, 374
Input Mask editor, 164
InputBox function, 289
InputQuery function, 289
InquireSoap interface, 897
InRange function, 89
Insert command for records, 520
Insert method, 128
insert operations, trigger firing from, 565
InsertComponent method, 118–119
InsertObjectDialog method, 481
InsertRecord method, 701
inside-out activation, 484
Install COM+ Object dialog box, 495–496
Install Into New Application page, 496
Installable Indexed Sequential Access Method (IISAM) drivers, 624
installing
  DLL wizards, 41
  property editors, **391–392**
instances, checking for, 307–310
InstanceSize method, 105
instancing in COM, **463**
Integer type, 478
Integrated Development Environment. *See* IDE (Integrated Development Environment)
Integrated Translation Environment (ITE), 368, 399
InterBase
  generators and triggers in, **565–566**
  history of, **562**
  IBConsole application, **562–564**, *563–564*
  IBX. *See* InterBase Express (IBX)
  operation of, **561–562**
  real-world examples
    booking classes, **608–611**, *610*
    case-insensitive searches, **602–603**
    free query forms, **613**, *613*
    generators and IDs, **600–602**, *601*
    locations and people, **603–605**
    lookup dialogs, **611–613**
    user interfaces, **605–608**, *607*
  server-side programming in, **564–565**
InterBase Express (IBX), **507**, **591**
  administrative components in, **593**
  dataset components in, **592–593**
  example, **593–594**
  live queries in, **594–597**, *598*
  monitoring, **598**, *599*
  system data in, **599–600**, *599*
InterBase Objects program, 564
InterBase Workbench tool, 564
InterceptGUID property, 652
interface IDs (IIDs), 459
interface section, 400
Interface Wizard, 432, *433*
Interfaced Component Reference model, 21
interfaces, 48, **69–71**
  and base forms, **330**, **333–334**
  for COM server, **461–463**, *462*, **465–466**
  for compound components, **354–356**
  of controls, 483
  in packages, **420–424**
  user. *See* user interface
InterfaceSupportsErrorInfo method, 458
InterlockedDecrement function, 458
InterlockedIncrement function, 458, 822
internal components in compound components, **349–350**
internal instancing, 463
internal messages, 302, **372–374**, *373*
internal objects in Automation, **482–483**, *483*
InternalAddRecord method
  in TMdCustomDataSet, 699
  in TMdDataSetOne, 701
InternalAfterOpen method
  in TMdCustomDataSet, 691–692
  in TMdDirDataset, 707
InternalCancel method, 712

InternalClose method, 694
InternalDelete method, 699
InternalEdit method, 712
InternalFieldDefs method, 710–711
InternalFirst method, 697
InternalGotoBookmark method, 697
InternalHandleException method, 703
InternalInitFieldDefs method
  in TMdCustomDataSet, 692–694
  in TMdDirDataset, 707–708
InternalInitRecord method, 699
InternalLast method, 698
InternalLoadCurrentRecord method
  in TMdDataSetStream, 700–701
  in TMdListDataSet, 706
internally calculated fields, 513
InternalOpen method, 690–691
InternalPost method, 699, 701
InternalPreOpen method
  in TMdCustomDataSet, 691
  in TMdListDataSet, 705
InternalRecordCount method
  in TMdCustomDataSet, 698–699
  in TMdListDataSet, 706
InternalRethinkHotkey method, 182
InternalScriptName property, 774
InternalSetToRecord method, 697
Internet, actions for, 224
Internet Direct (Indy) components, **738**, **741–744**, *744*
  HTML. *See* HTML (Hypertext Markup Language)
  HTTP. *See* HTTP (Hypertext Transfer Protocol)
  for mail, **748–750**, *749*
Internet Direct (Indy) project, 951
Internet Express, **858**
  JavaScript support in, **859–864**, *861*, *863*
  XMLBroker component in, **858**
Internet page, 7, 788
Internet programming, **737**
  HTML. *See* HTML (Hypertext Markup Language)
  HTTP. *See* HTTP (Hypertext Transfer Protocol)
  for mail, **748–750**, *749*
  with sockets. *See* socket programming
  XML. *See* XML
Internet Server API (ISAPI), **769–770**
InternetCloseHandle function, 755
InternetOpen function, 755
InternetOpenURL function, 755
InternetReadFile function, 755
Interop services, 908
interposer classes, **332–333**
IntfColSel unit, 421–422
IntfFormPack package, 421–422
IntfFormPack2 package, 421
IntfPack example, 421
IntfPack package, 421
INTO clause in SELECT, 628
InTransaction property, 589
IntraWeb library, **809–810**
  architectures in, **815**
  for Web applications, **810–815**, *812*
    building, **815–818**, *817*
    databases, **825–832**, *827–828*, *830*, *832*
    layout, **824–825**, *825*
    multipage, **818–821**, *819*
    sessions management, **821–823**, *823*
    with WebBroker, **823–824**
IntraWeb page, 810
IntToStr function, 85
INumber interface, 465
invalid areas, 266
Invalidate method, 266–267
InvalidateRect function, 267
InvalidateRegion function, 267
Invoke method, 468–469
IObjectContext interface, 497
IP addresses, **739**
IPictureDisp interface, 478
IProduceContent interface, 772
IProperty interface, 388
is operator, 67–68
ISAPI (Internet Server API), **769–770**
IsCallerInRole method, 497
IsConsole variable, 84
IsCursorOpen method, 695
IsDefaultPropertyValue function, 148
IsEqualGUID function, 87
IsInfinite function, 88
IsInTransaction method, 497
IsLibrary variable, 84
IsMultiThread variable, 84
IsNan function, 88
IsNull method, 703
isolation levels in transactions, 590
IsSecurityEnabled method, 497
IStrings interface, 478
IsUniDirectional function, 572

- ISupportErrorInfo interface, 458
- IsZero function, 89
- ITE (Integrated Translation Environment), 368, 399
- ItemHeight property, 187
- ItemIndex property, 250
  - in ListBox, 167
  - for radio group components, 259
- ItemProps property, 171
- Items property
  - in ActionBars, 246
  - for fonts, 343
  - for images, 189
  - in ListBox, 167
  - in TBookMarkStr, 542
  - in TList, 133
  - in TreeView, 194
- Items string list, 169
- ItemsEx property, 169
- IUnknown interface, 69, 84, **457–458**
  - class factories in, **460**
  - GUIDs in, **458–460**, *459*
- IVBSAXContentHandler interface, 851
- IVBSAXErrorHandler interface, 851
- IW Component Park, 810
- IWAppFormCreate method, 818, 827
- IWButton component, 811
- IWChart component, 810
- IWClientGrid example, 831–832, *832*
- IWClientSideDataSet component, 831
- IWClientSideDataSetDBLink component, 831
- IWCSLabel component, 831
- IWCSNavigator component, 831
- IWDataModulePool component, 810
- IWDialogs component, 810
- IWDynamicChart component, 831
- IWDynGrid component, 831
- IWEdit component, 811
- IWGranPrimo component, 810
- IWGridDemo example, 828–830, *828*, *830*
- IWLayoutMgrHTML component, 824
- IWListbox component, 811
- IWModuleController component, 823
- IWOpenSource component, 810
- IWPageProducer component, 823
- IWScrollData example, 826–827, *827*
- IWServerControllerBaseNewSession method, 822
- IWSession example, 821–823, *823*
- IWSimpleApp example, **811–815**, *812*
- IWTranslator component, 810
- IWTree example, 816–818, *817*
- IWTwoForms example, 818–821, *819*
- IXMLDocument interface, 839, 848
- IXMLDocumentAccess interface, 836
- IXMLNode interface, 839
- IXMLNodeCollection interface, 839
- IXMLNodeList interface, 839

J

- JavaScript
  - in Internet Express, **859–864**, *861*, *863*
  - for WebSnap, 791
- JavaScript language, 814
- JEDI (Joint Endeavor of Delphi Innovators), **952**
- Jet engine, **624**
  - for Excel, **625–626**, *626*
  - for importing and exporting, **628–629**
  - for Paradox, **624–625**, *625*
  - for text files, **626–628**
- JIT (Just in Time) compiler, **907–908**
- joins, **636–637**
- Joint Endeavor of Delphi Innovators (JEDI), **952**
- JPEG file format, 784
- JpegImage component, 783–784
- JScript, 791
- Just in Time (JIT) compiler, **907–908**

K

- KeepRowTogether property, 731
- keyboard
  - for form input, **259–261**, *260*
  - messages for, 377
- KeyField property, 519
- KeyOptions property, 172
- KeyPreview property, 259
- keys
  - in associative lists, 132–133
  - in database design, **558–560**
  - in keyset cursors, 631
- keyset cursors, **631**
- Kind property, 286
- KPreview example, 259–260, *260*
- Kylix, CLX in, 110

L

- LabelDoubleClick method, 285–286
- LabeledEdit component, **163**

LabelPosition property, 163
LabelSpacing property, 163
Language Exceptions page, 74
large documents with XML, **869–873**, *871*
LargeImages property, 189
LargeXML example, 869–871, *871*
LastSession property, 246
late binding
  abstract methods in, **66–67**
  for message handlers, **66**
  overriding and redefining methods in, **64–65**
  and polymorphism, **63–64**, *64*
  virtual vs. dynamic methods in, **65–66**
layout for Web applications, **824–825**, *825*
Layout Manager, 825, *825*
LayoutChanged method, 684–685
least-recently used menu items, **245–247**, *245*
Left property
  for components, 18
  for forms, 268
  in TControl, 159
Lesser General Public License (LGPL), 949
LessThanValue constant, 89
LGPL (Lesser General Public License), 949
LIB file, 401
LibComp example, 153–154
$LIBPREFIX directive, 407
libraries
  ActiveX, 34, 460–461
  for controls
    dual, **151–155**, *152*
    visual, **155–156**, *156*
  DataSnap, 649
  for Delphi for .NET Preview, **934–940**, *940*
  DLL. *See* DLLs (dynamic link libraries)
  extending, **337–340**
  in IDE, **5**
  IntraWeb. *See* IntraWeb library
  names for, **407–408**
  type, **468–470**
library path setting, 347
library statement, 403
LibSpeed example, 155
$LIBSUFFIX directive, 407
$LIBVERSION directive, 407
LIC files, 36
licenses
  for ActiveX controls, 488
  for author tools, **949**
  for Qt libraries, 150
line breaks in text files, 85
Lines class, 379
Lines string list, 164
Lines toolbar in Rave, 718
LineStart method, 147
LinkedActionLists property, 247
linking, 480
  to definitions, 13
  dynamic vs. static, **397–398**
  to Web database details, **827–830**, *828*, *830*
Linux
  compatibility functions for, 88
  libraries for, **155–156**, *156*
list actions, 224, **248–250**, *249*
list-based data-aware controls, **517–518**, *518*
list boxes, **167**, **186–188**
list components. *See* lists and list components
List Template Wizard, 945
ListActions example, 248–250, *249*
ListBox component, **167**
ListBoxForm form, 378–379
ListCli example, 478
ListControl property, 249
ListDate example, 132
ListDemo example, 129–131, *130*
ListDialDemo example, 381, *381*
ListDialog component, **393–395**, *394*
listening socket connections, 740
lists and list components, **128–131**, *130*
  CheckBoxList, **169**
  ComboBox, **168**
  ComboBoxEx and ColorBox, **169**
  as datasets, **705–706**
  hashed associative, **132–133**
  ListBox, **167**
  ListView and TreeView, **169–170**
  type-safe, **133–135**
  ValueListEditor, **170–172**, *170*
ListServ example, 478
ListSource property, 519
ListTest example, 387
ListView component, **169–170**, **188–193**, *191*
ListView Item Editor, 189, *191*
literals in input masks, 164
live queries in IBX, **594–597**, *598*
LIVE_SERVER_AT_DESIGN_TIME directive, 477

- loadable views, **16–18**, *17–18*
- LoadBalanced property, 667
- LoadDynaPackage method, 423
- LoadFile method, 322
- LoadFromFile method
  - in datasets, 510–511, 645
  - in TreeView, 194
  - in TStringList and TStrings, 128
- LoadFromStream method
  - in TBlobField, 139
  - in TStream, 137
- loading packages, **419–420**
- LoadLibrary function, 408
- LoadPackage function, 419
- LoadParamsOnConnect property, 572
- local domain names, **739**
- local tables for ClientDataSet
  - connections for, **509–510**, *510*
  - defining, **511–512**, *512*
- LocalConnection component, 653
- localhosts in IP addresses, 739
- Localizable property, 368
- Locate method, 514
- LocateFileService component, 788, 798
- locating
  - files, **798**
  - records, **514–515**
- locations
  - ADO cursor, **629–630**
  - in InterBase example, **603–605**
- Locked property, 481
- locking
  - control position, 19
  - records, 587
    - in ADO, **635–636**, **639–641**
    - in client/server programming, 557
    - recursive, 86
- LockType property, 635, 638
- Log Changes property, 515
- LogException method, 75–76
- logging
  - for ClientDataSet, 515
  - errors, **75–76**, *76*
- logical identifiers, 559
- logical keys, 558
- logical operators, 513
- logical three-tier architecture, 648
- LoginForm adapter, 795
- LoginFormAdapter component, 807
- LoginPrompt property, 572
- logins in WebSnap, **807–808**
- lookup controls, **519–520**, *519*
- lookup dialogs, **611–613**
- lookup fields, **533–534**, *534*
- ltBatchOptimistic value, 635, 638
- ltOptimistic value, 635
- ltPessimistic value, 635

M

- $M+ compiler directive, 114
- macros
  - for ModelMaker, **445–446**
  - shortcut keys for, 16
- mail, **748–750**, *749*
- Mail Merge Editor, 729
- MailMergeItems property, 729
- Main Form option, 41
- main forms in MDI applications, **316–317**, *316*
- Main window, looking for, **307–308**
- MainForm object, 301
- MainForm property, 277, 297
- MainInstance variable, 84
- maintained aggregates, 550
- maintenance releases of packages, 417
- major releases of packages, 417
- MakeObjectInstance method, 317
- MakeSplash method, 290–292
- MakeXmlAttribute function, 884–885
- MakeXmlStr function, 884
- managed code, 905, **908–909**
- manifest files for themes, 220–221
- manifests, 906
- ManualDock method, 235
- ManualFloat method, 235
- many-to-many relations, 557–558
- mapping
  - object-relational, 559
  - in XML, **854–857**, *854*, *856–857*, **886–887**, *887*
- Mapping page, 854
- MapTable example, 856–857, *857*
- MapViewOfFile function, 413–414
- markers in XML, 834–835
- markup languages, 833
- marshaling
  - in Automation, 467
  - in COM, 457

MaskEdit component, **163–164**
MastDet example, 552–553, *553*
master/detail relations
    in ClientDataSet, **552–553**, *553*
    in DataSnap, **664–665**, *665*
    in Rave reports, **730–731**, *731*
    in WebSnap, **803–805**, *805*
MasterAdapter property, 803
MasterDataView property, 731
MasterFields property, 552
Mastering Delphi Com+ Test example, 496
mastering.gdb database, 600
MasterKey property, 731
Math unit, **88–90**, *90*
MaxRecords property, 858
MaxRows property, 762
MaxSessions property, 805
MaxValue property, 658
MBParade example, 289
MDAC (Microsoft Data Access Components), **616–617**
MdComEvents library, 498
MdDbGrid component, 683, *683*
MdDesPk package, 392, 395
MdEdit1 example, 229–230
MdEdit2 example, 232–236, *232*, *236*
MdEdit3 example, 247–248
MDI (Multiple Document Interface) applications, **310**
    child forms and merging menus in, **315–316**, *315*
    frame and child windows in, **311–315**, *313*
    main forms in, **316–317**, *316*
    style property for, 254
    subclassing client windows in, **317–318**
MDI window actions, 224
MDIChildCount property, 312
MDIChildren property, 312, 316
MdiDemo example, **313–315**, *313*
MdiMulti example, 315–316, *315*
MdiMulti Demo4 example, 316–317, *316*
MdListBoxForm class, 378–379
MdListDialog class, 381
MdPack package, 347–348
MdWArrowImpl1 example, 488
Mediator pattern, 449
members, converting and moving, 447
Memo component, **723**
Memo controls, 836
memo fields in Rave, **729**
MemoRich component, **164–165**
memory
    allocation routines for, 84
    DLLs in, **411–414**, *415*
    garbage collection for, **911–916**
    for methods, 115
    for objects, **58–59**
    for screen, 265–266
    for strings, 58
    units for, **95–96**
memory leaks, 56
memory management records, 334–335
memory manager, **334–336**
memory-mapped files, sharing data with, **413–414**, *415*
Memory Snap tool, **948–949**
Menu Designer, **174**, *174*
menu.html file, 793
menus, 173
    accelerator keys for, **181–182**
    in control Bars, **234**
    images on, 185
    items on
        least-recently used, **245–247**, *245*
        owner-draw, **185–186**, *186*
    Menu Designer for, **174**, *174*
    merging
        in MDI applications, **315–316**, *315*
        in OLE, 481
    new, **7**
    pop-up, **174–176**
    in ToolBar, **216–217**
MergeChangesLog method, 515
merging menus
    in MDI applications, **315–316**, *315*
    in OLE, 481
message boxes, **289**
$MESSAGE directive, 44
message directive, 66
message keyword, 369
message-response methods, 369
message spy program, 33
MessageBox method, 289
MessageDlg function, 289
MessageDlgPos function, 289
messages and message handlers
    compiler, 31–32
    internal, **372–373**, *373*
    late binding for, **66**
    mail, **748–750**, *749*

- overriding, **368–372**
  - queues for, 302
  - in sequence diagrams, 433
  - for triggering events, 124
  - user-defined, **309–310**
  - in Windows, **303–304**
  - in windows controls, **373–378**
- metadata
  - in CLI, 904
  - in dbExpress, **578–579**, *579*
  - in Jet, 628
- Metafile component, **724**
- Method Implementation Code Editor, **441–442**, *442*
- method pointers for events, **125–126**
- MethodAddress method, 115
- MethodName method, 115
- methods
  - abstract, **66–67**
  - for ActiveX controls, 484
  - characteristics of, **46–47**
  - code completion for, 14–15
  - in DataSnap, **663–664**
  - in Delphi for .NET Preview, 925
  - messages for, 376
  - names for, 46
  - overloaded, **404**
  - overriding and redefining, **64–65**, 70
  - virtual, 70
    - in COM server, **466–467**
    - vs. dynamic, **65–66**
- MGA (Multi-Generational Architecture), 561
- Microsoft Access databases, 624
- Microsoft Data Access Components (MDAC), **616–617**
- Microsoft Database Engine (MSDE), 617
- Microsoft Indexing Service, 618
- Microsoft Intermediate Language (MSIL). *See* CIL (Common Intermediate Language)
- Microsoft.Jet driver, 617
- Microsoft libraries, **934–940**, *940*
- Microsoft Shared Source CLI, 905
- Microsoft Transaction Server (MTS), 494
- Microsoft Word
  - with Automation, 479
  - DLLs for, 411
- MIDAS (Middle-tier Distributed Application Services), 649
- Midas library for ClientDataSet, **510–511**
- MidasLib unit, 511, 569
- MilliSecondOfMonth function, 91
- mind-map diagrams, 435
- Minimize method, 269
- MiniSize program, 82–83
- MinSize property, 180–181
- MinValue property, 658
- mirroring in Rave, **732–733**
- MltGrid example, 543, *543*
- modal forms, **410–411**
- modal windows, 280
- ModalResult property, 285, 379
- modeless dialog boxes, 280, **285–287**, *285*
- ModelMaker, **429–430**
  - class diagrams in, **431–432**, *432–433*
  - coding features of, **435**
    - Delphi integration with, **436–437**
    - Difference view, **442–443**, *443*
    - Event Types view, **443**
    - managing, **438–439**, *439*
    - Method Implementation Code Editor, **441–442**, *442*
    - refactoring, **446–447**
    - templates for, **450–452**, *450*
    - Unit Code Editor, **439–441**
  - common diagram elements in, **435–436**, *436*
  - design patterns in, **447–450**
  - documentation in, **444–445**, *444*
  - features in, **452**
  - inheritance in, **437–438**
  - internal model in, **430**
  - macros for, **445–446**
  - non-UML diagrams in, **435**
  - sequence diagrams in, **433–434**, *433*
  - use case diagrams in, **434**
- Modified method
  - in TPropertyPage, 490
  - in TSoundProperty, 390
- Modified property, 313
- ModifyAccess property, 808
- ModuleIsPackage variable, 424
- modules
  - data modules, **535**, 648
    - for COM+, **497**
    - for WebSnap, **798–799**
  - System unit support for, 84
- Modules object, 793–794
- monitor resolution, 299

monitoring
  dbExpress connections, **576–577**, *577*
  IBX, **598**, *599*
Mono project, 905–906
mouse input
  dragging and drawing, **262–265**, *265*
  for forms, **261–262**
mouse messages, 374
mouse-related virtual keys, 262
MouseCoord method, 544
MouseOne example, 262–265, *265*
MoveBy method, 536
movement in datasets, **536–537**, **695–698**, *696*
  bookmarks for, **537–538**
  editing columns, **539–540**
  totals of columns, **537**, *537*
MoveTo method, 195, 212
moving
  classes, 446
  components, 18
  members, 447
MS-XML processor, 865
MSCorEE.dll file, 908
MSDAORA driver, 617
MSDAOSP driver, 617
MSDASQL driver, 617
MSDE (Microsoft Database Engine), 617
MSIL (Microsoft Intermediate Language).
  See CIL (Common Intermediate Language)
MSOLAP driver, 617
MSXML DOM, 871–872
MSXML library, 850–851
MSXML parser, 644
MSXML SDK, 838
MTS (Microsoft Transaction Server), 494
Multi-Generational Architecture (MGA), 561
multi-read synchronizer, 86
multi-threaded apartment model, 463
multicast events, **926–927**
MulticastDelegate class, 910–911, 927
Multiline Palette Manager, 947
multipage Web applications, **818–821**, *819*
Multiple Document Interface (MDI) applications, **310**
  child forms and merging menus in, **315–316**, *315*
  frame and child windows in, **311–315**, *313*
  main forms in, **316–317**, *316*
  style property for, 254
  subclassing client windows in, **317–318**
multiple frames without pages, **328–330**, *329*
multiple instancing, 463
multiple-page forms, **202**
  image viewer in, **207–210**, *207*
  PageControl and TabSheet for, **202–207**, *204–205*
  for user interface wizard, **210–213**, *211*
multiple pages, WebSnap, **789–791**, *790*
multiple selections
  DBGrid for, **542–543**, *543*
  in ListBox, 167
MultipleTransactionsSupported property, 590
multipurpose WebModule, **774–776**
MultiSect property, 167
multitasking, 302, **304**
multithreading, **304–307**
  class for, 86
  with socket connections, **738**
multitier databases. *See* DataSnap
Multitier page, 653
MultiWin example, 281–282
mutexes (mutual exclusion objects), **308**
MyBase file, **508**. *See also* ClientDataSet component
MyBase1 example, 509–510, *510*
MyBase2 example, 511–513, *512*
MySQL drivers, 567

N

name directive, 402
name mangling, 401–402
name-value pairs for lists, **129**
names and Name property
  accessing properties by, **115–117**, *117*
  for action objects, 222
  in C++, 401–402
  for components, 339
  components without, 122
  for domains, 739
  for fields, 360
  for fonts, 21
  for functions, 360
  for methods, 46, 115
  in ModelMaker code, 446
  for projects and library, **407–408**
  for properties, 360
  in TComponent, **121**
  in TField, 527
  for units, 44
namespaces
  in .NET architecture, 906
  for units, **922–924**

- namespaces clause, 923
- NameValues example, 170–171, *170*
- NameValueSeparator property, **129**
- native formats
  - in ClientDataSet, **511**
  - date time, 709
- native look-and-feel in library selection, 155
- navigating datasets, **536–537**, **695–698**, *696*
  - bookmarks for, **537–538**
  - editing columns, **539–540**
  - totals of columns, **537**, *537*
- NegInfinity constant, 88
- nested markers, 834
- nested transactions, **634–635**
- nested try blocks, 72
- nested types, **926**
- .NET architecture, **500–502**, *502*, **899**
  - assemblies in, **906–907**
  - CIL, **907–911**
  - CLI in, **903–904**
  - CLR in, **905–906**
  - Delphi for .NET Preview. *See* Delphi for .NET Preview
  - deployment and versioning in, **916–917**, *917*
  - garbage collection in, **911–916**
- .NET Framework, 900
- NetClassInfo example, 932–933, *933*
- NetCLX, 110, 112
- NetEuroConv example, 933
- NetImport example, 500–502, *502*
- NetLibrary library, 500–502
- NetLibSpeed example, 933
- NetNumberClass unit, 501
- Netscape Server API (NSAPI), **769–770**
- Neutral threading model, 463
- New Component dialog box, 341, *341*
- New Field dialog box, 531, *531*, 534
- New Form option, 41
- New Items dialog box, 39–40, *40*
  - for ActiveX controls, 489
  - for COM+, 494
  - for DataSnap, 653
  - for form inheritance, 319–320, *319*
  - for IntraWeb, 810
  - for Web services, 877, 889
  - for WebBroker, 770
  - for WebSnap, 787
  - for XML Data Binding Wizard, 847
- New Project option, 41
- New Transactional Object dialog box, 495, *495*
- New Web Server Application dialog box, 770, 773, 781
- New WebSnap Application dialog box, 787, *787*
- New WebSnap Page Module dialog box, 789, *790*
- NewButtonClick method, 301
- NewDate example, 59–60, *60*
- NewGuid example, 459–460, *459*
- NewInstance class, 335
- NewItem function, 174
- NewLine function, 174
- NewMenu function, 174
- NewPopupMenu function, 174
- NewSubMenu function, 174
- NewValue property, 641
- NewWinProcedure method, 318
- nil value
  - for forms, 282
  - with Free, 55, 58–59
- NoAction object, 227
- NodeIndentStr property, 842
- nodes
  - in TreeView, **197–198**, *199*
  - in XSLT, 864–865
- NodeType property, 840
- NonAware example, 545–547, *545*
- nonblocking socket connections, **738**
- nonvisual components, 111, **378–381**, *381*
- nonwindowed controls, 158
- normalization rules, **558**
- Notebook component, 202
- Notification method
  - for data-aware controls, 673
  - in TComponent, 301–302
  - in TMdPersonalData, 354
  - in TMdViewer, 355–356
- notifications
  - messages for, 374
  - in windows controls, **376–377**
- NotifyDataLinks method, 670
- NoTitle example, 258–259, *258*
- NSAPI (Netscape Server API), **769–770**
- null values with field events, **535–536**, *536*
- NullAsStringValue variable, 94
- NullDates example, 535–536, *536*
- NullEqualityRule variable, 94
- NullMagnitudeRule variable, 94
- NullStrictConvert variable, 94
- Numeric edit box
  - creating, **368–370**
  - with thousands separators, **370–371**

O

OBJ files, 36
ObjAddRef method, 458
ObjAuto unit, 148
ObjClass property, 710
ObjDataSetDemo example, 713, *713*
Object Debugger, **948**
object identifiers (OIDs)
  in database design, **558–560**
  in Interbase, **600–602**, *601*
Object Inspector, **19–21**
  for components, 351–352, *352*
  drop-down fonts in, **21**
  Object TreeView in, **22–23**
  property categories in, **21–22**
Object Inspector Font Wizard, 945
object linking, 480
Object Linking and Embedding (OLE), **456–457**
object-oriented programming (OOP), 43, 64
  and modeling. *See* ModelMaker
  for reusable components, 483
object reference model, **55–56**, *55*
  assigning objects in, **56–58**, *56*
  memory for objects in, **58–59**
object-relational mapping, 559
Object Repository, **39–41**, *40*
Object TreeView, **22–23**
ObjectBinaryToText method, 140
ObjectPropertiesDialog method, 482
objects
  assigning, **56–58**, *56*
  in Automation, **476–477**, **482–483**, *483*
  code completion for, 14–15
  for COM server, **461–463**, *462*
  in CTS, **909–910**
  datasets of, **710–713**, *713*
  in DataSnap, **667**
  in Delphi, **44–48**
  field, **527–528**, *528*
  memory for, **58–59**
  scope of, **476–477**
ObjectTextToBinary method, 140
ObjQueryInterface method, 458
ObjRelease method, 458
OBJREPOS subdirectory, 41
ObjsLeft example, 335–336
Observer pattern, 449
octets in IP addresses, 739
OCX (OLE Controls), 456, 483
OCX files, 36
ODBC (Open Database Connectivity), 615, 644
ODList example, 187–188
ODMenu example, 185–186, *186*
Office programs with Automation, **478–479**
oh.exe tool, 33
OIDs (object identifiers)
  in database design, **558–560**
  in Interbase, **600–602**, *601*
OLAP (Online Analytical Processing), 617
OldCreateOrder property, 153, 279
OldValue property, 641–642
OLE (Object Linking and Embedding), **456–457**
OLE Automation. *See* Automation
OLE container messages, 376
OLE Controls (OCX), 456, 483
OLE DB Provider, 617–618
OLE DB Provider Development Toolkit, 618
OLE DB Provider For Indexing Service, 617
OLE DB Provider For Internet Publishing, 617
OLE DB Provider For OLAP, 617
OLE DB providers, **616–617**
OLE Insert Object dialog box, 480–481
OleCont example, 482, *482*
OleContainer component, **480–482**, *482*
OnAction event, 771–772
OnActionExecute event, 228, 296
OnActionUpdate event, 296
OnActivate event, 278, 296, 298
OnActiveFormChange event, 299–300
OnAdvancedDrawItem event, 186
OnArrowDblClick event, 364–365
OnBeforeDispatchPage event, 790
OnBeforeExecuteAction event, 797
OnBeforePageDispatch event, 794
OnCalcFields event, 531, 585
OnCanResize event, 270
OnCanViewPage event, 808
OnChange event
  in TabControl, 208
  in TDate, 126
  in TMdArrow, 363
OnClick event, 125, 261
OnClose event, 278–279, 300
OnCloseQuery event, 278–279
OnCloseUp event, 168
OnColumnClick event, 191
OnCommand event, 742

OnCommandGet event, 757, 765
OnConstrainedResize event, 270
OnContextMenu event, 175
OnContextPopup event, 176
OnCreate event, 277–279, 299
OnCreateNodesClass event, 197
OnDataChange event, 546
OnDeactivate event, 296, 298
OnDeleteError event, 553
OnDestroy event, 279, 301
OnDockDrop event, 236
OnDockOver event, 235–236
OnDoubleClick event, 285
OnDragDrop event, 98
  for docking, 234
  for TreeView, 195
OnDragOver event, 98
  for docking, 234
  for TreeView, 195
OnDrawColumnCell event, 540–541
OnDrawItem event, 185
OnDrawTab event, 209
OnDropDown event, 168
one-to-many relations, 557
one-to-one relations, 557
OneCopy example, 308–310
OnEditButtonClick event, 533
OnEditError event, 553
OnEndDock event, 234, 236
OnEnter event, 177–178
OnException event, 75, 296, 553
OnExecute event
  in ActionFont, 313
  in ActionLink, 223–224
  for actions, 228, *228*
OnExecuteAction, 223
OnExit event, 177–178
OnFind event, 288
OnFindIncludeFile event, 798
OnFindStream event, 798
OnFindTemplateFile event, 798
OnFormatCell event, 763
OnGetData event, 668
OnGetDataSetProperties event, 667
OnGetItem event, 249
OnGetItemCount event, 249
OnGetResponse event, 864
OnGetText event, 535, 541, 549
OnGetValue event, 794, 797
OnHelp event, 257, 296
OnHint event, 217, 296
OnHTMLTag event, 760
OnIdle event, 215, 296, 304
OnItemSelected event, 250
OnKeyPress event, 163, 259
Online Analytical Processing (OLAP), 617
OnLogTrace event, 576
OnMeasureItem event, 185, 187
OnMessage event, 296
OnMinimize event, 296
OnMouseDown event, 261
OnMouseMove event, 261–262
OnMouseUp event, 261
OnMouseWheel event, 261
OnMouseWheelDown event, 261
OnMouseWheelUp event, 261
OnMove event, 181
OnNewRecord event, 547
OnPageAccessDenied event, 808
OnPaint event, 265–266, 274
OnPostError event, 553
OnReceivingData event, 893
OnReconcileError event, 587, 651, 659–660
OnRecordChangeComplete event, 641
OnRecordSetCreate event, 622
OnReplace event, 288
OnResize event, 269–271
OnRestore event, 296
OnScroll event, 271
OnSetText event, 535, 541
OnShortCut event, 296
OnShow event, 278
OnShowHint event, 183, 296
OnStartDock event, 234
OnStateChange event, 525, 546
OnStatusTextChange event, 486
OnTag event, 760
  in PageProducer, 779
  in WebSnap, 791
OnTimer event, 492
OnTitleChange event, 486
OnTitleClick event, 830
OnTrace event, 576
OnUpdate event
  in ActionCount, 227
  in ActionFont, 313

  and actions, 224, 229
  in MDI applications, 314
OnUpdateData event, 660
OnUpdateError event, 553, 641, 660
OnValidate event, 535
OnWillChangeRecord event, 641
OOP (object-oriented programming), 43, 64
  and modeling. *See* ModelMaker
  for reusable components, 483
OOP Form Wizard, 945
opaque strings, 538
Open Database Connectivity (ODBC), 615, 644
open-source projects, 951–952
Open Tools API, 452
Open XML project, 951
OpenDialog component, 287
OpenHelp tool, 33
opening datasets, **690–694**
opening markers in XML, 834–835
openItemClick method, 937, 939–940
OpenPictureDialog component, 288
OpenSchema method, 622–623, *623*
OpenTools API, 388
OpenXML DOM, 838
Operator property, 734
optimistic locking, 587
  in ADO, **639–641**
  in client/server programming, 557
options, components for, **165–166**
Options property
  in DataSetProvider, 658
  in DBGrid, 516
Oracle drivers, 567
orthogonal lines in ModelMaker, 452
out-of-process servers, 457
outer transactions, 634–635
OutLabel property, 353
Output Options dialog box, 720, *720*
overload directive, 404
overload keyword, 46, 65
overloaded constructors, 54–55
overloaded methods, 46, **404**
override keyword, 64
overriding
  Finalize, **911–912**
  message handlers, **368–372**
  methods
    in late binding, **64–65**
    virtual, 70
owner components, 58
owner-draw controls, **184–185**
  list box of colors, **186–188**
  menu items, **185–186**, *186*
  messages for, 377
Owner property, 119–120
OwnerDraw property, 185, 209
ownership in TComponent, **117–120**, *120*
OwnHandle property, 162
OwnsObjects property, 131

P

P/Invoke (Platform Invocation Service), 904
pa attributes, 388
~PA files, 37
Package Collection Editor tool, 33
Package Editor, 345, *345*, 347–348, *349*, 417, *417*
PackageInfo pointer type, 84
packages, **415–416**
  for components, **337–339**
  for Fonts combo box, **344–347**, *345–346*
  forms in, **417–419**, *417*
  interfaces in, **420–424**
  loading at run time, **419–420**
  and program size, 30–31
  structure of, **424–427**, *425*
  System unit support for, 84
  versioning in, **416–417**
Packages page, 346
PackAuto package, 477
PacketRecords property, 575, 584
packets
  with ClientDataSet, **584**
  in DataSnap, **651–652**, **667–668**
PackInfo example, 425–427, *425*
PackWithForm package, 418, 420
Page mode in IntraWeb, 815
page numbers in Rave, **724**
Page object, 793
PageControl component, 119, **202–207**, *204–205*
  actions for, 224
  docking to, **239–241**, *241*
  in XML, 836
PageControl property, 206
PagedAdapter component, 794
PageDesigner page, **717**
PageDispatcher component, 787, 794, 808
PageHead component, 774

- PageNumInit component in Rave, **724**
- PageProducer component, 759–761, 774, 779, 788
- pages
  - frames with, **326–328**, *327*
  - frames without, **328–330**, *329*
  - Web, ActiveX controls in, **492–493**, *493*
- Pages example, 202–207, *204–205*
- Pages list, 793
- PageScroller component, **172–173**
- PageSize property, 799
- PageTail component, 774–775
- Paint method, 358, **361–363**, *362*
- painting
  - DBGrids, **540–542**, *542*
  - in windows, **265–267**
- Panel component, 119
- Panels property, 382
- Paradox, Jet engine for, **624–625**, *625*
- ParamByName method, 581
- ParamCount global variable, 84
- parameters
  - code, **15**
  - for DLLs, 399
  - for mouse events, **262**
  - in overloaded methods, 46
  - in stored procedures, 565
- parametric queries
  - in DataSnap, **663**, *663*
  - in dbExpress, **579–581**, *580–581*
- Params property
  - in queries, 663
  - in SQLConnection, 572
  - in SQLDataSet, 580–581
  - in XMLBroker, 858
- ParamStr global variable, 84
- Parent property, **158–159**
- ParentColor property, 160
- ParentFont property, 160
- ParentXxx properties, 374
- ParQuery example, 580–581, *581*
- parsers, MSXML, 644
- parseURL method, 851
- PAS files, 36
- Pascal language, 38, 44
- passing XML documents, **884–886**
- PasswordChar property, 163
- Paste Link operation, 480
- PasteSpecialClick event, 481
- PasteSpecialDialog method, 481
- pasting components, **24–25**
- patch files, 415
- path delimiter functions, 87
- PathInfo property, 863
- patterns in ModelMaker, **447–450**
- PCE.exe tool, 33
- PChar type, 405–406
- PE (Portable Executable) format, 904, 908
- Peek method, 131
- people in InterBase example, **603–605**
- performance in InterBase, 561
- Performance Monitor for connections, 644
- permissions in WebSnap, **808**
- Persisted Recordset provider, 618
- persistency
  - in ADO recordsets, **644–645**
  - in Arrow component properties, **362–364**, *364*
  - class for, 113
  - in streaming, **140–142**, *141*
- Personal edition, 4
- pessimistic locking in ADO, **635–636**
- PEVerify utility, 909
- physical keys, 558–559
- PickList property, 516
- PixelsPerInch property, 275–276
- plain component packages, 338
- plain forms, **252–253**, *253*
- Platform Invocation Service (P/Invoke), 904
- platform-specific symbols, 156
- platforms for databases, **567–569**
- playing compound objects, 480
- PlaySound function, 64, 372, 391
- PMdRecInfo pointer type, 696
- po options, 666–667
- pointers
  - for bookmarks, 696
  - in Delphi for .NET Preview, 920
  - in object reference model, 55, *55*
  - for procedures, 409–410
- PoliForm example, 322–324, *322*
- PolyAnimals example, 64, *64*
- polymorphic forms, **321–324**, *322*
- polymorphism, **63–66**, *64*
  - abstract methods in, 66–67
  - in interfaces, 70
- pooling
  - connection, **643–644**
  - object, **667**
- Pop method, 131

- pop-up menus, **173–176**
- POP3 (Post Office Protocol, version 3), 740
- PopulateTypes method, 937–938
- Popup method, 175
- PopupActionBarEx component, 242
- PopupMenu property, 174–175
- portability
  - of database engines, 506
  - in library selection, 155
  - of text DFM files, 19
- Portable Executable (PE) format, 904, 908
- PortableDragTree example, 196–197
- porting programs, **247–248**
- ports in socket programming, **739–740**
- Pos function, 93
- PosEx function, 93
- position and Position property
  - of controls, **159**, 163
  - of Form Designer components, 18
  - of forms, 255, **268–269**
  - of mouse, 262
  - of scroll bars, 271
  - in TStream, 136
- Post command, 520
- POST method, 778
- Post Office Protocol, version 3 (POP3), 740
- PostMessage method, 66, 303–304, 310
- predefined items
  - actions, **224–225**
  - dialog boxes, **287–289**, *288*
- preemptive multitasking, 302
- Preferences page
  - for AutoSave feature, 11
  - for compiler, 30
  - for docking, 6
- prefixes
  - for functions, 360
  - for names, 121
- pressed-down state, 173
- Preview compiler, 900
- Preview pages, HTML, 788
- previewing in Rave, 719, *719*
- primary keys, **558–560**
- printing in dbExpress, **581–584**
- PrintOutDataSet method, 582–583
- PrioritySchedule property, 246
- private specifier, **48–50**, 925
- ProcessMessages function, 303–304
- Producer object, 793
- Producer property, 772, 776, 863
- ProducerContent property, 772
- Professional Studio edition, 4
- program flow in exceptions, **72–73**
- program statement, 922
- programmability, client/server programming for, 557
- ProgressBar component, **172**, **671–673**, *673*
- Project JEDI, **952**
- Project Manager, **26–28**, *27*, 723
- Project Options dialog box, 28
  - for applications, 296
  - for Automation, 477
  - for forms, 277, *278*, 280
  - for messages, 31, *31*
  - for ModelMaker documentation, 445
  - for packages, 346, *346*
- Project toolbar in Rave, 718
- Project Tree panel, **717**
- project-wide to-do items, 9
- ProjectFile property, 719
- projects, **26–28**, *27*
  - compiling and building, **30–32**
  - exploring, **32**, *32*
  - names for, **407–408**
  - options for, **28–30**
- prologues in XML, 835
- PropCom example, 466
- properties
  - accessing by name, **115–117**, *117*
  - for ActiveX controls, **489–491**, *490*
  - categories
    - in Object Inspector, **21–22**
    - registering, **366–367**, *367*
  - code completion for, 14–15
  - collections for, **381–385**, *383*
  - in CTS, **909–910**
  - in Delphi for .NET Preview, 925–926
  - editors
    - installing, **391–392**
    - for sound, **388–391**, *390–391*
    - writing, **388**
  - encapsulation with, **50–52**, *51*
  - events as, **126–128**
  - for forms, **53–54**, *53*
  - names for, 360
  - in Object Inspector, 20–21
- Properties dialog box, 20
- Property Editor dialog box, 438–439, *439*
- property keyword, 53

- Property panel, **717**
- protected specifier, **48–50**, **60–62**, 925
- protection, client/server programming for, 557
- Protection example, 61–62
- protocols
    - for DataSnap, **650–651**
    - in socket programming, **740**
- provider options in DataSnap, **666–667**
- ProviderFlags property, 658
- ProviderName property, 655
- proxy forms, 114
- PtInRegion function, 365
- public deployment, 916
- public specifier, **48–50**, 52
- published keyword, 52, **114–115**
- Publisher example, 499–500
- publishing
    - subcomponents, **350–352**, *352*
    - WSDL, **881–883**, *881*
- pull-down menus, 173
- purely abstract classes, 69
- Push method, 131
- PWideChar type, 460

Q

- QBIcons example, 257
- QDynaForm example, 252
- QLibComp example, 153–154, 156, *156*
- QLibSpeed example, 155
- QMbParade example, 289
- QMouseOne example, 262–263
- QMultiWin example, 281–282
- QRefList example, 192–193
- QRefList2 example, 283
- QRegion class, 366
- QScale example, 275–276
- QStdCtrls unit, 152
- Qt library, 150
- queries
    - live, **594–597**, *598*
    - parametric, **663**, *663*
    - WebBroker technology for, **777–781**, *780*
- Query component, 507
- query forms, **613**, *613*
- QueryFields property, 777–778
- QueryInterface method
    - for as casts, 465
    - in IUnknown, 457–458
- QueryTableProducer component, 759, 777, 779
- queues, 132, 302

R

- radio items on menus, 173
- RadioButton component, **165–166**
- RadioGroup component, **166**
- RadioItem property, 173, 230
- raise keyword, 71
- raising exceptions, 71
- RandomFrom function, 89
- RandomRange function, 89
- Range property, 270, 272–273
- ranges, components for, **172–173**
- Rave, **715–716**
    - calculations in, **733–734**
    - components of, **722–723**
        - basic, **723–725**
        - data access, **725–726**, *725*
        - data-aware, **728–730**, *729*
        - regions and bands, **726–728**, *727*
    - data connections in, **721–722**, *722*
    - design environment in, **716–719**, *716*
    - master/detail reports in, **730–731**, *731*
    - mirroring in, **732–733**
    - rendering formats for, **720–721**, *720*
    - RVProject component in, **719–720**, *719*
    - scripts for, **731–732**, *732*
- RaveDatabase component, 726
- RaveDetails example, 730–732, *731–732*
- RaveDirectDataView component, 726
- RaveDriverDataView component, 726
- RaveLookupSecurity component, 726
- RaveProject component, 723
- RaveSimpleSecurity component, 726
- RaveSingle example, 721–722, *722*
- RC files, 37
- RDSConnection component, 508, 619
- re-raising exceptions, 71
- read clauses, 50, 52, 465
- Read method, 143
- read-only ProgressBars, **671–673**, *673*
- read-only properties
    - in Object Inspector, 20
    - setting, 52
- read-write TrackBar, **674–677**, *677*
- ReadBool method, 332
- ReadBuffer method, 137

ReadComponent method, 136
ReadComponentRes method, 140
ReadInteger method, 332
ReadString method, 332
Real48 type, 920
ReallocMem function, 921
rebuild.bat file, 900
Rebuild Wizard, 946
RecNo property, 537
Reconc unit, 588
Reconcile Error Dialog unit, 588, *588*
ReconcileProducer property, 860
ReconcileProvider property, 863
record buffers, **698–703**
record viewer component, **678–683**, *682*
RecordChanged method, 679
RecordCount property, 537, 560, **631–632**
recording macros, shortcut keys for, 16
records
  in Delphi for .NET Preview, 920–921
  editing, 520
  locating, **514–515**
  locking, 587
    in ADO, **635–636**, **639–641**
    in client/server programming, 557
    recursive, 86
  status of, **585**, *585*
recordsets in ADO
  clone, 633
  disconnected, **642–643**
  persistent, **644–645**
recursive locks, 86
redefining methods, **64–65**
refactoring ModelMaker code, **446–447**
reference-counting techniques, 476
references, class, **76–78**, *78*
referential integrity, **559**
ReflectionForm class, 935
Reflector tool, 901
RefList example, 189–193, *191*
RefList2 example, 283–285, *284*
Refresh in Model option, 438
Refresh method, 266, 660
refreshing
  data, **660–662**, *661*
  screen, 266–267, 660
RefreshRecords method, 660–661
regasm (Framework Assembly Registration Utility), 501
region components, **726–728**, *727*
Register ActiveX Server command, 488
Register method
  for component editors, 395
  for components, 342, 357
  for packages, 345
  for property categories, **366–367**
  for property editors, 391
RegisterActions method, 387
RegisterClass method, 330
RegisterColorSelect method, 421
registering
  ActiveX libraries, 34, 488
  Automation servers, **473–474**
  component editors, **395**
  components, 342
  conversion rates, 99–101
  property categories, **366–367**, *367*
RegisterPooled method, 667
RegisterPropertyInCategory function, 366–367
Registry
  for automation servers, 473–474
  cleaning up, 33
  for COM servers, 464
  for connection pooling, 644
  for GUIDs, 458–459
  for hot-track activation, 24
  and INI files, **331–332**
  for shared component templates, 26
  for sound properties, 389–390
  status information in, 39
RegSvr32.exe program, 464
relational databases, 560
relations
  in database design, **557–558**
  master/detail. *See* master/detail relations
relationships in diagrams, 435
Release method, 292, 820
_Release method, 71, 457
ReleaseCapture function, 265
Remote Data Module Wizard, 654, *654*
RemoteServer component, 649
RemoteServer property, 655
Remove method
  in Delphi for .NET Preview, 926–927
  for lists, 128
RemoveComponent method, 119
removing
  captions, 258
  form fields, **121–122**

renaming in ModelMaker code, 446
rendering formats for Rave, **720–721**, *720*
Repaint method, 266–267
RepaintRequest method, 363
reparenting classes, 446
Repeatable Read transaction isolation mode, 561, 590
replicable data-aware controls, **674**
ReplyNormal property, 741
Report Library node, 717
reporting
  in dbExpress, **581–584**
  dynamic database, **776–777**
  in Rave. *See* Rave
request methods in WebBroker, 778
Request object, 793
Request property, 771
Requires a New Transaction option, 495
Requires a Transaction option, 494
requires keyword, 346
requires lists, 345–346
RES files, 37
reserved characters in XML, 834
resize cursors, 255
ResizeStyle property, 181
resolution of monitors, 299
resolvers, 660
ResolveToDataSet property, 658, 660
Resource Explorer tool, 34
Resource Workshop tool, 34
resources and resource files
  binding into executables, 30
  in COM+, 494
  DLLs for, 399
  Windows, 27–28
resourcestring keyword, 85
Response object, 793
Response property, 771
Restore method, 269
Result pages, HTML, 788
Resume method, 752
Resync method, 642
RethinkHotkeys method, 182
RethinkLines method, 182
return values in stored procedures, 565
reusability
  design patterns for, **448–449**
  object-oriented programming for, 64, 483
Revert to Inherited command, 319
RGB color, 160–161
RGB function, 160–161
RichBar example, **214–215**, *214*, 217–219, *218*
RichEdit actions, 224
RichEdit component, **164–165**
RichEditSelectionChange method, 214–215, 229
right mouse button, 261
robustness diagrams, 435
Roeder, Lutz, 901
role-based security, 494
Rollback action, 596
Rollback method, 589
RollbackRetaining command, 597
RollbackTrans method, 634
rolling back transactions, 589, 634–635
Rotor project, 905–906
Round function, 89–90
rounding
  in currency conversions, **101–102**
  numbers, **89–90**, *90*
Rounding example, 90, *90*
RoundTo function, 89–90
Row property, 167
RowAttributes property, 763
RowCurrentColor property, 826–827
RowHeights property, 683
RowHeightsChanged method, 683
RowLayout property, 167
RowLimit property, 826
RPS files, 37
RTL (run-time library) package, **81**, 83, **110–112**, *110*
  converting data, **96–102**, *98*, *101*
  file management, **102–104**, *103–104*
  TObject, **104–108**, *106*, *108*
  units of. *See* units
RTTI (run-time type information)
  for casting, **67–69**
  for SOAP, 878
Run method, 296
run-only component packages, 338
Run Parameters dialog box, 42, 773
run time
  calling DLLs at, **408–410**
  loading packages at, **419–420**
run-time library for Delphi for .NET Preview, **930–931**
run-time library (RTL) package, **81**, 83, **110–112**, *110*
  converting data, **96–102**, *98*, *101*
  file management, **102–104**, *103–104*
  TObject, **104–108**, *106*, *108*
  units of. *See* units

run-time packages, 30–31, 338
run-time only properties, 52
run-time type information (RTTI)
    for casting, **67–69**
    for SOAP, 878
RunningTotal property, 730
RunProp example, 116–117, *117*
RvCustomConnection component, 721
RvDataSetConnection component, 721
RvNDR Writer component, 720
RVProject component, **719–720**, *719*
RvQueryConnection component, 721
RvRenderHTML component, 721
RvRenderPDF component, 720
RvRenderPreview component, 720
RvRenderPrinter component, 720
RvRenderRTF component, 721
RvRenderText component, 721
RvSystem Writer component, 720
RvTableConnection component, 721
RWBlocks example, 601, **605–613**, *607*, *610*

S

safe code, **908–909**
safecall calling convention, 471, 664
SafeLoadLibrary function, 408
SameValue function, 89
sample1embedded.xml file, 865–866
SampProv driver, 617
Save Code Template dialog box, 451
Save method in MDI applications, 314
SaveDialog component, 288
SavePictureDialog component, 288
SavePoint property, 515
SaveStatusForm unit, 331, 333
SaveToFile method, 128, 510–511, 645
SaveToStream method
    in TBlobField, 139
    in TStream, 137
saving
    application status information, 331–332
    compiler options, 28–29
    desktop settings, **5–6**
    DFM files, 19
    docking status, 238
SAX (Simple API for XML), **850–853**, *853*, 872–873
SaxDemo1 example, 850–853, *853*, 872
scalability and cursor location, 630
Scale example, 275–276
ScaleBy method, 275–276
Scaled property, 275–276
scaling forms, **274–277**
schema information
    in ADO, **622–623**, *623*
    in Jet, 628
    in XML, **849–850**
SCHEMA.INI file, 627
SchemaObject parameter, 578
SchemaPattern parameter, 578
SchemaTest example, 579, *579*
SchemaType parameter, 578
ScktSrvr.exe application, 650
scope in Automation, **476–477**
screen
    refreshing, 266–267, 660
    size of, 299
Screen example, 299–302, *300*
Screen object, **299–302**, *300*
ScreenSnap property, 269
scripts
    for adapters, **797–798**
    for Rave, **731–732**, *732*
    server-side, **791–793**, *792*
Scroll1 example, 271–272, *272*
Scroll2 example, 273–274, *273*
ScrollBar component, **172**
ScrollBox component, **173**
scrolling forms, **270–271**
    automatic, **272–273**
    coordinates in, **273–274**, *273*
    example, **271–272**, *272*
    messages for, 377
SDI (Single Document Interface), 310
sealed keyword, **924–925**
search actions, 224
search application for HTTP, **751–754**
search engines, **785–786**, *786*
search path setting, 347
search results, **9–10**
search utility, 33
searches, case-insensitive, **602–603**
searching window lists, **308–309**
second-level message handlers, 369
secondary forms, **280**
    adding, **280**
    creating, **281–282**
SecondaryShortCut property, 222

SecondOfWeek function, 91
Section component, **723–724**
security
    client/server programming for, 557
    in COM+, 494, 496
    in Web pages, 493
SelAttributes property, 164
SelCount property, 167
SELECT statement, 628–629
SelectDirectory function, 103
Selected property, 167
SelectedColor property, 419
SelectedRows property, 542
selecting components
    in Form Designer, 18
    referenced by properties, 20
SelectNextPage method, 204
Self keyword, 46–47
self pointer, 70, 125
SelItem property, 379
semicolons (;)
    for connection strings, 620
    in SELECT, 628
Sender object, 228
sending data
    database requests, **547–549**, *548*
    to Excel tables, 479
    mail, **748**
    over socket connections, **744–747**, *747*
    to Word, 479
SendList program, 748, *749*
SendMessage function, 304
SendResponse method, 784
SendStream method, 784
SendToDb example, 548–549, *548*
separators for toolbars, 213
sequences
    diagrams for, **433–434**, *433*
    in Oracle, 565
server constraints, **657–658**
server-side cursors, 629, 636
server-side programming, **564–565**
server-side scripts, **791–793**, *792*
server-side support components for DataSnap, **653**
server socket connections, 740
ServerAddress property, 745
ServerGUID property, 655
ServerName property, 655
servers
    Automation, **470**, *470*
        code for, **472–473**
        in components, **477–478**, *477*
        registering, **473–474**
        type-library editor for, **470–472**, *471*
    COM. *See* COM (Component Object Model)
service providers, 618
Services file, 740
Session component, 793
SessionCount property, 246
sessions
    in Web applications, **821–823**, *823*
    in WebSnap, **805–806**, *807*
SessionService component, 805–806
Set methods in TAXForm1, 491
set operations on lists, 128
Set_Value method, 473
SetAbort method, 497
SetBookmarkData method, 696–697
SetBookmarkFlag method, 696
SetBounds method
    in TControl, 159
    in TMdArrow, 361
    in TMdRecordView, 680
SetCapture function, 264
SetChangeFormFont method, 344
SetComplete method, 497
SetData method, 414, 702
SetDataField method
    in TMdDbProgress, 672–673
    in TMdDbTrack, 676
SetDataSource method
    in TMdDbProgress, 672
    in TMdDbTrack, 676
SetDirection method, 359–360
SetFieldData method
    in TMdDataSetOne, 702–703
    in TMdDirDataset, 706
SetFirstName method, 353
SetForegroundWindow function, 307, 309
SetLabel method, 354
SetLines method, 379–380
SetLinesPerRow method, 684
SetMemoryManager function, 84, 334
SetMoreData method, 384
SetOleFont method, 478
SetOleStrings function, 478

SetOptionalParameter method, 668
SetPen method, 363
SetPropValue method, 419
SetRecNo method, 698
SetRoundMode function, 90
SetSchemaInfo method, **578–579**, *579*
SetShareData method, 414
SetSubComponent method, 351, 357–358
SetText method, 356
SetTextLineBreakStyle function, 85
setTimeout function, 823
Settings page, 836
SetValue method
    in TDate, 51
    in TSoundProperty, 390
SetViewer method, 356
SetWindowLong function, 298, 317
SetWindowOrgEx function, 274
SetYear method, 51
SharedConnection component, 653
ShareMem unit, **95–96**, 405–406, 409
sharing data with memory-mapped files, **413–414**, *415*
ShellExecute function, 212, 750
shift-state modifiers, 262
shortcut keys, **16**
ShortCut property, 222
Show Compiler Progress option, 30
Show Component Captions setting, 19
Show method, 280, 286, 820
ShowApp example, 297–298
ShowColumnHeaders property, 191
ShowException method, 75–76
ShowForm method, 606
ShowFrame method, 329
ShowHints property, 183
ShowInfoProc function, 426–427
Showing property, 159
ShowMainForm property, 297
ShowMessage function, 289, 403
ShowMessageFmt method, 289
ShowMessagePos method, 289
ShowModal method, 280, 282, 285, 411
ShowSender method, 106
ShowStatus method, 751, 754
ShowStringForm method, 252
ShowTotal method, 305–306
Sign function, 89
signals in Qt, 162
Simple API for XML (SAX), **850–853**, *853*, 872–873
Simple Mail Transfer Protocol (SMTP), 740
Simple Object Access Protocol (SOAP), **876**, **889–892**
Simple Security Controller option, 726
SimpleDataSet component, **573–574**, 576
SimpleMemTest unit, 335–336
SimpleObjectBroker component, **667**
SimplePanel property, 217
SimpleRoundTo function, 90
SimpleText property, 217, 751
Single Document Interface (SDI), 310
single-instance secondary forms, **281–282**
single instancing, 463
single page access rights, **808**
single-threaded Apartment model, 463
Single threading model, 463
Single type, 478
size
    components, 19
    controls, **159**
    executables, **82–83**
    fonts, 276–277
    forms, **269**
    screen, 299
Size property, 136
sLineBreak constant, 85
SmallImages property, 189
SmallInt type, 478
Smith-Ferrier, Guy, 615
SMTP (Simple Mail Transfer Protocol), 740
SMTP component, 748
SN SDK, 916
Snap To Grid option, 18
SnapBuffer property, 269
snapping to screen, **269**
SOAP (Simple Object Access Protocol), 651, **876**, **889–892**
SOAP-based solutions, 651
SOAP headers, debugging, **887–888**, *888*
Soap Server Data Module, 889
SoapConnection component, 889
SoapDataClient example, 891
SoapDataServer example, 890
SoapEmployee example, 886–887, *887*
SoapEmployeeIntf unit, 883
SOAPServerIID property, 891
socket programming, **737–739**
    connections in, **740**
    for database data, **744–747**, *747*

- domain names in, **739**
- high-level protocols in, **740**
- Indy components in, **741–744**, *744*
- IP addresses in, **739**
- ports in, **739–740**

SocketConnection component, 652
sort by directive, 573
SortFieldNames property, 573
sorting in ListView, 191
SortType property, 191
Sound button, **371–372**
sound properties editor, **388–392**, *390–391*
SoundDown property, 388
soundex algorithm, 91–92
SoundUp property, 388
source code files, 27–28, **38–39**
Source Doc Generation page, 445
Source Doc Import page, 445
Source Options page, 10, 15
special characters in XML, 834
special keys, messages for, 376
speed
- in Automation, **476**
- in library selection, 155

Splash example, *290*, 292
splash screens, **289–292**, *290*
Splash1 example, 290–291
Splash2 example, 291–292
Split1 example, 180, *180*
Split2 example, 181
SplitH example, 181
SplitLong method, 786
Splitter component, 180
splitting forms, **180–181**, *180*
SQL Links drivers, 507
SQL Monitor tool, 33
SQLClientDataSet component, 574
SQLConnection component, **569–572**, *571*, 575, 589
SQLDataSet component, 573, 803
SqlMon.exe tool, 33
SQLMonitor component, **574**, **576–577**, *577*
SQLOLEDB driver, 617
SQLTimeStampToStr function, 574
stacks, 131–132, 398
Standalone mode in IntraWeb, 815
Starkey, Jim, 561–562
startDocument event, 850, 852
startElement event, 850, 852, 872
StartID property, 830
StartTransaction method, 589–590
state diagrams, 434
state of windows, 268–269
State property
- in CheckBoxList, 169
- in datasets, **524–525**, 701

state-setter commands, 173
StateImages property, 189
static binding, 63
static cursors, **630–631**
static linking, **397–398**
static members, **925–926**
static methods, 70
StaticListAction component, 248–249
StaticSynchronize method, 305
Statistics page, 773
status
- of ClientDataSet records, **585**, *585*
- of datasets, **524–525**

status bars, **217–219**, *218*, **718–719**
StatusBar component, 217
StatusFilter property, 586
stdcall calling convention
- for DLLs, 399
- for Web services, 880

stdcall directive, 402
StdConvs unit, 91, 96
StdCtrls unit, 152
StdVCL library, 478
Stop on Delphi Exceptions options, 74
store and paint approach, 266
stored directive, 344
stored procedures, **564–565**
StoreDefs property, 511
StoredProc component, 507
StoredProcName property, 573
StoreRAV property, 719
Str function, 83
StrDemo example, 92
StreamDSDemo example, 703–704, *704*
streaming, **135**
- classes for, **137–138**
  - compressing, **145–146**, *145*
  - custom, **142–145**
  - TReader and TWriter, **139–140**
  - TStream, **135–137**
- vs. code generation, **113–114**
- files, **138–139**
- persistency in, **140–142**, *141*

string lists
    in Automation, **478**
    for combo boxes, 169
    for CSS, 797
    for memo lines, 164
    for properties, 52
StringFromGuid2 function, 459
StringReplace method, 512
strings
    comparisons with, 91–92
    conversions with, 86–87, 187–188
    in Delphi for .NET Preview, **920–921**
    exporting from DLLs, **404–406**
    for lists, **128–129**
    memory for, 58
    opaque, 538
    unit for, 91–93
    URL, 752
    in XML, 837
StringToColor function, 188
StringToFloatSkipping method, 370–371
StringToGUID function, 459
StringValuesList component, 794
StripParamQuotes property, 760
strong name assemblies, 916
StrToBool function, 86
StrUtils unit, **91–93**
stub routines, 70
style sheets, **764–765**, 797, 859
StyleRule property, 797, 859
styles and Style property
    in Application, 219
    for CLX, **219–220**, *220*
    in ColorBox, 169
    in ComboBox, 168
    for forms, **254**
        borders, **254–256**, *254*
        windows, **257–259**, *258*
    in InetXPageProducer, 859
    for list boxes, 187
    in ModelMaker, 452
    in TMdFontCombo, 343
    in TToolButton, 213
Styles string list, 797
StylesDemo program, 219–220, *220*
StylesFile property, 797, 859
subclassing, **317–318**
subcomponents, publishing, **350–352**, *352*
SubMenuImages property, 174
Support function, 87
SupportCallbacks property, 652
SupportedBrowsers property, 814
Supports function, 424
Supports transactions option, 495
surrogate keys, 559
symbols
    code completion for, 15
    defining, 156
    in diagrams, 435
synchronization classes, 86
Synchronize method
    in TFindWebThread, 751
    in TThread, 305–306
SyncObjs unit, 148
Syntax Helper option, 488
syntax of XML, **834–835**
SysConst unit, **85–86**
SysInit unit, **84–85**
system colors, 160
system data in IBX, **599–600**, *599*
system-level interfaces, 615
system tables, 578
System unit, **84–85**
SysUtils unit, **85–88**, **102–104**, *103–104*

T

tab actions, 224
TabbedNotebook component, 202
TabBmp object, 210
TabControl component, 202
    for image viewer, 207–210, *207*
    for pages, 328–329, *329*
table bookmarks, 538
Table component, 507
TableAttributes property, 763
TableName property
    in datasets, 573, 704
    in TADOTable, 624
tables
    constraints on, **657–658**
    editing columns in, **539–540**
    events for, **658–659**
    in HTML, **762–764**, *763*
    system, 578
    totals of columns in, **537**, *537*
    virtual method tables, 66, 70, 466
TabOrder property, 176

TAboutBox class, 290
tabs in editor, 11
Tabs menu, 24
TabSet component, 202
TabSheet components, **202–207**, *204–205*
TabStop property, 176
TabVisible property, 205, 210
TAction class, 222, 385
TActionBar collection, 244
TActionClient class, 244
TActiveXControl class, 458, 488
TActiveXControlFactory class, 460
TADTField class, 529
Tag property, **124**
TAggregateField class, 529
TagParams list, 761
tags in XML, 833
TAlignment class, 230
TAnimal class, 62–64, 67
TApplication class, 295–296
TAppServerOne class, 654
TargetList function, 386–387
TArrayField class, 529
TAutoIncField class, 529
TAutoObject class, 458
TAutoObjectFactory class, 460
TAXForm1 class, 491–492
TBasicAction class, 222
TBCDField class, 529
TBinaryField class, 529
TBitBtn class, 184, 373
TBitmap class, 208
TBits class, 147
TBlobField class, 139, 529
TBlobStream class, 138
TBookmark class, 538
TBookmarkList class, 542
TBookmarkStr class, 542–543
TBooleanField class, 529
TBrush class, 358
TBucketList class, 132
TButton class, 152, 173
TBytesField class, 529
TCanTest class, 384
TCanvas class, 150, 262–263
TCGIApplication class, 771
TCharProperty class, 388
TCheckConstraint class, 657
TClass class, 77
TClassFinder class, 147
TClassList class, 131
TClipboard class, 208
TCollection class, 131, 147
TCollectionItem class, 131, 147, 382
TColor class, 160–161
TComObject class, 457–458, 464
TComObjectFactory class, 460, 462
TComponent class, 110–111, *110*, **117**, 340
  in Delphi for .NET Preview, 931–932
  form fields in
    hiding, **122–123**
    removing, **121–122**
  Name property in, **121**
  ownership in, **117–120**, *120*
  Tag property in, **124**
TComponentEditor class, **393**
TComponentList class, 131
TCompressStream class, 145
TConnectionAdmin class, 579
TContainedAction class, 222
TControl class, **158**
  activation and visibility, **159**
  colors, **160–161**
  fonts, **159–160**
  Parent property, **158–159**
  size and position, **159**
TCoolBand class, 231
TCoolBar component, 231
TCP/IP (Transmission Control Protocol/Internet Protocol), 652, 737–739
TCP/IP sockets, 651
TCP ports, **739–740**
TcpClient class, 737–738
TcpServer class, 737–738
TCriticalSection class, 305
TCurrencyField class, 529
TCustomAction class, 222
TCustomActionToolBar class, 247–248
TCustomADODataSet, 618
TCustomConnection class, 569
TCustomControl class, 158, 487
TCustomDockForm class, 234
TCustomEdit class, 369
TCustomForm class, 251–252
TCustomGrid class, 679
TCustomLabel class, 350

TCustomMemoryStream class, 137
TCustomSQLDataSet class, 572
TCustomVariantType class, 94
TCustomWebDispatcher class, 771
TDataLink class, **669–670**
TDataModule class, 147, 535, 653
TDataSaxHandler class, 872–873
TDataSet class, **520–524**, 536, 552, 592, 687
TDataSetField class, 529
TDatasetProvider class, 658
TDate class, **51**, *51*
TDateField class, 529
TDateForm class, 56–58
TDateListI class, 134–135
TDateListW class, 134–135
TDateTime class, 49–50
  in DateUtils unit, 91
  in Delphi for .NET Preview, 921
  in System unit, 84
TDateTimeField class, 529
TDBGHack class, 544
TDecompressStream class, 145
TDump.exe tool, 33
TeamSource tool, 33
Telnet ports, 740
temperature conversions, 96
templates
  code, **15**, **450–452**, *450*
  in Component Palette, **25–26**, *26*
  HTML, 788
  in Object Repository, 40–41
  in UDDI, 895
TEncodedStream class, 142–143
TEnumProperty class, 388
TerminateAndRedirect method, 820
TestCom example, 464–465
testing
  COM server, **464–466**
  datasets, **703–704**, *704*
  debugger for, 774
TestStreamFormat method, 140
TEvent class, 305
text
  for case diagrams, 434
  DFM files stored as, 19
  in Rave, **723**
text-based data-aware controls, **516**
Text component, **723**
text files
  Jet engine for, **626–628**
  line breaks in, 85
Text IISAM, 626–627
text-input components
  Edit, **163**
  LabeledEdit, **163**
  MaskEdit, **163–164**
  MemoRich and RichEdit, **164–165**
  TextViewer, **165**, *165*
Text property
  in ComboBox, 168
  in DOM, 844
  in Edit, 162
  in MemoRich, 164
  for panels, 217
TextBrowser control, 756
TextHeight attribute, 187
TextViewer component, **165**, *165*
TField class, 525–527, 641–642
TFieldDataLink class, 670–671
TFileData class, 707
TFiler class, 139, 147
TFileRec class, 85
TFileStream class, 137
TFindWebThread class, 752–754
TFirstServer class, 473, 477
TFloatField class, 527, 529
TFMTBCDField class, 529
TForm class, **251–252**, 333
  for border icons, **256–257**, *257*
  for form style, **254**
    borders, **254–256**, *254*
    windows, **257–259**, *258*
  for plain forms, **252–253**, *253*
TFormBitmap class, 331, 333–334
TFormBorderStyle class, 256
TFormClass class, 77
TFormScrollBar class, 270
TFormSimpleColor class, 422
TFrameClass class, 329–330
TFrameList class, 324–325
TGraphicControl class, 158, 340, 359, 487
TGraphicField class, 530
TGUID class, 458–459
TGuidField class, 530
THandleStream class, 137
THashedStringList class, 133

themes, **220–221**, *222*
thick frame border style, 255
thin clients, 649
ThinCli1 example, 656
ThinCli2 example, 659, *659*
ThinPlus example, 662–665, *663*, *665*
THintInfo structure, 183
THome class, 788–789
thousands separators, **370–371**, 533
ThousandSeparator character, 533
threading methods in COM, **463**
threads, 84, **304–307**
threadvar variables, 822
three-tier architecture, 648
tiDirtyRead value, 590
TIDispatchField class, 530
TidThreadSafeInteger class, 822
Tile method, 312
TileMode property, 312
time
    native formats for, 709
    unit for, 91
Timer class, 350–351
Timer property, 351
timers
    for messages, 302
    in Web applications, 823
    XClock, **492–493**, *493*
TimeStamp field, **574**
timing, 476
TInformSubscriber class, 498–499
TIniFile class, **331–332**
TIntegerField class, 530
TIntegerProperty class, 388
TInterfacedField class, 530
TInterfacedObject class, 70, 84, 457
TInterfacedPersistent class, 147
TInterfaceList class, 147
TInvokableClass class, 880
tiReadCommitted value, 590
tiRepeatableRead value, 590
TISAPIRequest class, 771
TISAPIResponse class, 771
TItemProp class, 171
Title property, 379
titles for applications, 297
TIWServerController class, 822
TJpegImage class, 783
TLargeIntField class, 530
TLB (type library) file, 37, 470
TLibCli example, 474–475
TlibdemoLib_TLB file, 477
tlibimp utility, 501
TList class, **128–131**, *130*, 133
TListControlAction class, 385
TListItems class, 189
TMdActiveButton class, 372–373
TMdArrow class, 359–360, 363–365
TMdClock class, 350–351
TMdCustomDataSet class, 687–689
TMdCustomListAction class, 385–386
TMdDataSetStream class, 689–690
TMdDbGrid class, 684
TMdDbProgress class, 671–672
TMdDbTrack class, 675
TMdDirDataset class, 706
TMdFontCombo class, 341–344
TMdListBoxDialog class, 379
TMdListCompEditor class, 393–395
TMdListCopyAction class, 386
TMdListCutAction class, 386
TMdListDataSet class, 705
TMdListPasteAction class, 386
TMdMyColleciton class, 382–383
TMdMyItem class, 382
TMdNumEdit class, 369
TMdObjDataSet class, 710
TMdPersonalData class, 352–354
TMdRecInfo class, 695–696
TMdRecordLink class, 678
TMdRecordView class, 679
TMdSoundButton class, 371–372, 392
TMdThousandEdit class, 370–371
TMdViewer class, 355–356
TMdWArrowX class, 488
TMemoField class, 530
TMemoryManager class, 84, 334–335
TMemoryStream class, 137–138
TMethod class, 115, 125
TMultiReadExclusiveWriteSynchronizer class, 86
TMySaxHandler class, 851–852
TMySimpleSaxHandler class, 852
TNewDate class, 59–60
TNotifyEvent type, 364
TNumber class, 462, 464
TNumericField class, 530

to-do lists, **7–9**, *8*
TObject class, 54, 84, **104–108**, *106*, *108*
TObjectBucketList class, 132
TObjectField class, 530
TObjectHelper class, 930
TObjectList class, 131–133
TObjectQueue class, 132
TObjectStack class, 131–132
TODO comments, **7–9**
TODO files, 37
TOleStream class, 138
toolbars and ToolBar control, **213**, **230–231**
  actions with, *224*, **229–230**
  captions for, 236
  ControlBar, **231–234**, *232*
  CoolBar, **230–231**
  docking in, **235–239**, *236*
  hints in, **182–184**, *184*, 218, *218*, 236
  menus and combo boxes in, **216–217**
  in Rave, **718**
  RichBar example, **214–215**, *214*
Tools API, 388
Tooltip symbol insight feature, 12–13
tooltips, **182–184**, *184*
  expression evaluation, 16
  hints, 19
Top property
  for components, 18
  for forms, 268
  in TControl, 159
Total example, 537, *537*
totals
  of columns, **537**, *537*
  in Rave, **729–730**
TOwnedCollection class, 147
TParser class, 147
TPen class, 358
TPersistent class, 110, **112–113**
  for Arrow component, **362–364**, *364*
  property access in, **115–117**, *117*
  published keyword for, **114–115**
TPoint structure, 93
TPromptQueryButton class, 859
TPropertyEditor class, 388
TPropertyPage class, 490
TPropertyPage1 class, 490
TQueue class, 132
TrackBar component, **172**, **674–677**, *677*
tracking forms, **299–302**, *300*
TrackMode property, 815
training classes, **608–611**, *610*
transactional data modules, **497**
TransactionLevel property, 591
transactions
  actions for, 596
  in ADO, **634–636**
  in client/server programming, **589–591**, *591*
  in COM+, 494
  InterBase for, 561
TransactionsSupported property, 590
TranSample example, 591, *591*
transformations
  XML, **854–857**, *854*, *856–857*, **886–887**, *887*
  XSL, **868–869**
TransformNode method, 866, 868
TransformProvider component, 855, *856*
translations, BabelFish, **876–879**, *877*, *879*
Transmission Control Protocol/Internet Protocol (TCP/IP), 652, 737–739
transparent color in forms, **267**, *267*
TransparentColor property, 267
TransparentColorValue property, 267
TReader class, **139–140**, 147
TRecall class, 147
TReconcileAction class, 588
TReconcileErrorForm dialog box, 641
TRect structure, 93
trees in XML, 835
TreeView, **22–23**
TreeView component, **169–170**, **193–197**, *194*
  custom nodes in, **197–198**, *199*
  for XML documents, **839–841**, *840*
TreeView Items property editor, 193
TReferenceField class, 530
TRegIniFile class, **332**
TRegistry class, **332**
TRegSvr.exe example, 464
TRegSvr.exe tool, 34
TRemoteDataModule class, 654
TResourceStream class, 137
triangle symbol in Band Style Editor, 728
triggered events, 124
triggers in Interbase 6, **565–566**
trigonometric functions, 89
Trunc function, 921
try blocks, 71–73

TryEncodeDate function, 87
TryEncodeTime function, 87
TryFinally example, 72–73
TryStrToCurr function, 87
TryStrToDate function, 87
TryStrToFloat function, 87
TSaveStatusForm class, 331, 334
TSchemaInfo class, 622
TScreen class, 299
TScrollingWidget class, 252
TScrollingWinControl class, 252
TSetProperty class, 388
TSingleEvent class, 305
TSmallIntField class, 530
TSmallPoint class, 93
TSOAPAttachment class, 893
TSoapDataModule class, 890
TSoapTestDm class, 890
TSoundProperty class, 389
TSQLConnection class, 569
TSQLDataSet component, 572, 578
TSQLTimeStampField class, 530, **574**
TStack class, 131–132
TStream class, **135–137**
TStringField class, 530
TStringHash class, 133
TStringList class, **128–129**, 148
TStringProperty class, 388
TStrings class, **128–129**
TStringStream class, 137
TTags class, 761
TTextRec class, 85
TThread class, 147, 305
TThreadList class, 147
TTimeField class, 530
TToolButton class, 173, 213
TTransactionDesc type, 590
TTreeItems class, 197
TTreeNode class, 195
TTypedComObject class, 458
TTypedComObjectFactory class, 460
tuples, 560
Turbo Grep tool, 33
Turbo Register Server tool, 34
TUserSession class, 821–822, 825–826
TVarBytesField class, 530
TVarData record class, 84
TVariantField class, 530
TVariantManager class, 84
TViewerForm class, 322–323
TWebAppPageModule class, 787
TWebAppPageModuleFactory class, 807
TWebBrowser class, 485
TWebContext class, 806
TWebPageModuleFactory class, 807
TWebRequest class, 771, 775
TWebResponse class, 771
TWideStringField class, 530
TWidgetControl class, 158, **161–162**, 251
TWinControl class, 158, **161**, 251, 340
TWindowArrange class, 312
TWindowCascade class, 312
TWindowClose class, 312
TWindowMinimizeAll class, 312
TWindowTileHorizontal class, 312
TWindowTileVertical class, 312
TWinSocketStream class, 138
TWordField class, 530
TWriter class, **139–140**, 147
TXActAttributes class, 635
type libraries
    in Automation, **468–470**
    editors for, **470–472**, *471*
type library (TLB) file, 37, 470
Type Library Importer, 501
Type Library page, 471
type-safe containers and lists, **133–135**
type-safe down-casting, **67–69**
typed parameters, 565
TypeInfo unit, 116–117
types
    checking, 458
    COM, **478**
    in Delphi for .NET Preview, **920**, **926**
    in inheritance, **62–63**
Types unit, **93**
TypesList method, 882
TypInfo unit, 148

U

UDDI (Universal Description, Discovery, and Integration) specification, **894–898**, *896*
UDDIBrow.ini file, 895
UddiInquiry example, 895–898, *896*
UDL files, 37

- UML diagrams, 429–430
  - class, **431–432**, *432–433*
  - common elements in, **435–436**, *436*
  - sequence, **433–434**, *433*
  - use case, **434**
- UndoLastChange method, 514
- Unicode character sets, 835
- unidirectional cursors, **560–561**, **566–567**
- UniPrint example, 582–584
- Unit Code Editor, **439–441**
- unit dependency diagrams, 435
- units
  - code editors for, **439–441**
  - in dbExpress, **568–569**
  - in inheritance, 60–61
  - names for, 44
  - namespaces for, **922–924**
  - in Project Explorer, 32
  - in RTL, **81–82**
    - COM-related, 96
    - ConvUtils and StdConvs, 91
    - DateUtils, **91**
    - DelphiMM and ShareMem, **95–96**
    - Math, **88–90**, *90*
    - StrUtils, **91–93**
    - SysConst, **85–88**
    - System and SysInit, **84–85**
    - SysUtils, **85–88**, **102–104**, *103–104*
    - Types, **93**
    - Variants and VarUtils, **94–95**, *94*
  - in VCL and CLX, 157
- Universal Description, Discovery, and Integration (UDDI) specification, **894–898**, *896*
- UnloadPackage function, 419–420
- unmanaged code, 905
- unsafe casts, 44
- unsafe code, 44, 909
- unsafe types, 44, 920
- Update Criteria property, 640
- Update method
  - in painting, 266
  - in TMdMyColleciton, 383
- update regions, 267
- UpdateBatch method, 638
- UpdateClock method, 352
- UpdateData method, 677
- UpdateLabel method, 353
- UpdateList method, 135
- UpdateMode property, 640
- UpdateOleObject method, 490
- UpdateRegistry method
  - in DataSnap, 667
  - in TRemoteDataModule, 654
- updates
  - in ADO, 615
    - batch, **638–642**
    - conflicts in, **641–642**
    - joins, **636–637**
  - in ClientDataSet, **584–589**, *585–586*, *588*
  - in DataSnap example, **659–662**
  - in dbExpress, **576–578**
  - DLLs, 416
  - triggers fired by, 565
- UpdatesPending property, 638
- UpdateStatus method, 585, 638
- UpdateTarget method
  - in TAction, 385
  - in TMdCustomListAction, 385–386
- UpDown component, **172**
- uppercase names, converting to, 602–603
- upsizing applications, 556
- URL strings, 752
- URLs
  - in IntraWeb, 814–815
  - in WinInet, 755
- UsageCount property, 246
- use cases, diagrams for, **434**
- UseCol example, 411
- UseDockManager property, 235, 240
- UseFrame property, 826
- UseMem example, 413–414, *415*
- user-defined window messages, **309–310**
- user interface
  - ActionList component for, **222–224**, *223*
    - example, **225–228**, *226*, *228*
    - predefined actions in, **224–225**
    - with toolbars, **229–230**
  - ActionManager architecture, **241–250**, *243–245*, *249*
  - CLX styles, **219–220**, *220*
  - docking support, **234–241**, *236*
  - in InterBase, **605–608**, *607*
  - multiple-page forms. *See* multiple-page forms
  - themes, **220–221**, *222*
  - toolbars, **213–219**, **230–234**, *232*
  - for wizard, **210–213**, *211*
- user logins, **807–808**

user-related adapters, 794
uses statement
    for controls, **154**
    in Delphi for .NET Preview, 923–924
    for external symbols, 15
    in implementation section, 256
UseSOAPAdapter property, 891

V

validation in XML, **849–850**
Value property, 525
ValueChecked property, 517
ValueFromIndex property, **129**
ValueListEditor component, **170–172**, *170*
ValueUnchecked property, 517
VarArrayCreate function, 514
VarArrayOf function, 514
VarComplexCreate functions, 95
varEmpty type, 477
variables
    code completion for, 14–15
    pointers to, 55, *55*
variant records, 921
Variant type, 468, 478
    custom, 94–95, *94*
    System unit support for, 84
Variants unit, **94–95**, *94*
varNull type, 477
VarSQLTimeStampCreate function, 574
VarUtils unit, **94–95**, *94*
VBX standard, 483
VCL (Visual Components Library), **110–112**, *110*
VCL controls
    converting, **156–158**
    for Delphi for .NET Preview, **931–933**, *933*
    dual libraries support for, **151–155**, *152*
    visual libraries for, **155–156**, *156*
    vs. VisualCLX, **149–158**, *152*
VCL.DCP file, 408
VCL Hierarchy Wizard, 946–947
VCL package, 83
VCL70.BPL file, 408
VclToClx tool, 158, **947–948**
verbs
    in compound documents, 480
    in IComponentEditor, 393
version tags in DCU files, 416
versions
    dbExpress, **568–569**
    DLL, 399
    in InterBase, 561
    in .NET architecture, **916–917**, *917*
    in packages, **416–417**
    TeamSource control system, 33
VertScrollBar property, 270
ViewAccess property, 808
ViewGrid example, 682, *682*
views, loadable, **16–18**, *17–18*
virtual action items, 250
virtual destructors, 55
virtual fields, 50
virtual functions, 64–65
virtual keys, 262
virtual keyword, 64–65
virtual method tables (VMTs), 66, 70, 466
virtual methods, 398
    in COM server, **466–467**
    vs. dynamic, **65–66**
    in interfaces, 70
VirtualListAction component, 248–249
VirtualListAction1GetItem method, 250
visibility and Visible property
    for actions, 222
    for controls, **159**
    in Delphi for .NET Preview, **925**
    for fields, 658
    for inherited forms, 321
    for scroll bars, 270
    for secondary forms, 280
    in TControl, 159
    in TField, 528
VisibleButtons property, 516
VisibleRowCount property, 685
Visitor pattern, 449
Visual Basic controls, 483
Visual Components Library (VCL), **110–112**, *110*
visual editing, 484
visual form inheritance, **318–319**
    from base forms, **319–321**, *319–320*
    polymorphic, **321–324**, *322*
visual libraries, **155–156**, *156*
visual styles in ModelMaker, 452
VisualCLX controls, 110–111
    converting, **156–158**
    dual libraries support for, **151–155**, *152*

  - vs. VCL, **149–158**, *152*
  - visual libraries for, **155–156**, *156*
- VMTs (virtual method tables), 66, 70, 466
- Voice method, 63–64
- vtables, 66, 70, 466

W

- W3C (World Wide Web Consortium)
  - for HTML, 759
  - for XML, 833
- WaitForSingleObject function, 308
- $WARN directive, 44
- warnings, compiler, 31–32, 44
- Watch List, 42
- Web and Web pages. *See also* Internet programming
  - actions for, 224
  - ActiveX controls in, **492–493**, *493*
  - hit counter for, **782–785**
  - search engines for, **785–786**, *786*
  - services for, **875–876**
    - attachments, **892–893**, *894*
    - BabelFish translations, **876–879**, *877*, *879*
    - currency conversion, **880–882**, *881*, *883*
    - database data, **883–887**, *887*
    - debugging SOAP headers, **887–888**, *888*
    - existing classes as, **888–889**
    - SOAP and WSDL, **876**
    - UDDI, **894–898**, *896*
  - WebSnap for. *See* WebSnap application
- Web App Debugger tool, 33, **772–774**, *774*, **887–889**, *888*
- Web Services Description Language (WSDL), **876**, **881–883**, *881*
- Web Services page, 889
- WebAppComponents component, 787
- WebAppDbg.exe tool, 33, **772–774**, *774*
- WebApplication class, 817
- WebBroker technology, **770–772**
  - with Apache, **781–872**
  - debugging in, **772–774**, *774*
  - for dynamic database reporting, **776–777**, *777*
  - with IntraWeb, **823–824**
  - for multipurpose WebModule, **774–776**
  - for queries and forms, **777–781**, *780*
  - Web hit counter, **782–785**
  - Web searches, **785–786**, *786*
- WebBrowser control, **485–486**, *486*
- WebConnection component, 652
- WebContext object, 806
- WebDemo example, 485–486, *486*
- WebDispatch property, 858
- WebDispatcher component, 771–772
- WebFind example, 751–754, *755*
- WebModules, 770–772
- WebRequestHandler object, 773
- WebSearcher example, 785–786, *786*
- WebSnap application, **786–789**, *787*
  - for adapters, **794–798**, *796*
  - for databases, **798**
    - data editing, **801–803**, *802*
    - data module for, **798–799**
    - DataSetAdapter for, **799–801**, *800*
    - master/detail in, **803–805**, *805*
  - with IntraWeb, **823–824**
  - logins in, **807–808**
  - for multiple pages, **789–791**, *790*
  - permissions in, **808**
  - for server-side scripts, **791–793**, *792*
  - sessions in, **805–806**, *807*
  - XSLT with, **866–868**, *866*
- WebSnap page, 787
- well-formed XML, **835**
- What's this? Help, 256
- wheel events, 261
- White Pages in UDDI, 894
- white space in XML, 834
- wide string support, 87
- WideCompareStr function, 87
- WideCompareText function, 87
- WideFormat function, 87
- WideLowerCase function, 87
- WideSameStr function, 87
- WideSameText function, 87
- WideString type, 87, 459–460, 478
- WideUpperCase function, 87
- Width property
  - for components, 18
  - for forms, 268–269
  - in Splitter, 180
  - in TControl, 159
  - in TMdArrow, 359
- WINAPI modifier, 401
- window lists, searching, **308–309**
- window procedures, 161
- WindowMenu property, 311
- WindowProc property, 161

- windows, 161
  - controls in, 158, **368**
    - message handlers in, **368–373**, *373*
    - messages in, **373–378**
    - notifications in, **376–377**
    - overriding message handlers in, **371–372**
  - for forms, **257–259**, *258*
  - painting in, **265–267**
- Windows Interactive SQL (WISQL), 562
- Windows menu, 7
- Windows operating system
  - background processing and multitasking in, **304**
  - common dialogs, **287–288**, *288*
  - events in, **302–303**
  - messages in, **303–304**, **309–310**, 374
  - resource files from, 28
  - themes in, **220–221**, *222*
- WindowState property, 268–269
- WindowsXP.res file, 220
- WinInet API, **754–756**
- WinShoes component, 739
- WinSight tool, 33, 303
- WinVersion example, 85–86
- WISQL (Windows Interactive SQL), 562
- WithinPastDays function, 91
- wizards
  - DLL, 41
  - user interface for, **210–213**, *211*
- WizardUI example, 210–213, *211*
- wm_Char message, 368
- wm_Command message, 376
- wm_EraseBkgnd message, 317
- wm_HScroll message, 272
- wm_LButtonDblClk message, 365
- wm_LButtonDown message, 371
- wm_LButtonUp message, 371
- wm_NCCalcSize message, 269
- wm_NCHitTest message, 258, 269
- wm_NCLButtonDown message, 269
- wm_NCPaint message, 269
- wm_Size message, 270
- wm_User message, 66, 301
- WMHScroll method, 272
- WNDCLASS, 307
- WndProc method, 161, 317
- Word
  - with Automation, 479
  - DLLs for, 411
- WordBool type, 478
- WORDCALL.TXT file, 411
- WordCont example, 482–483, *483*
- World Wide Web. *See* Internet programming; IntraWeb library; Web and Web pages
- World Wide Web Consortium (W3C)
  - for HTML, 759
  - for XML, 833
- Wrapper pattern, 449
- wrapper properties, 350
- wrappers
  - for ActiveX controls, 485
  - for XML, 838
- write clauses, 50, 52, 465
- Write method, 143
- write-only properties, 52
- WriteBool method, 332
- WriteBuffer method, 137
- WriteComponent method, 136, 140, 142
- WriteComponentRes method, 140
- WriteInteger method, 332
- WriteString method, 332
- ws constants, 269
- Ws.exe tool, 33, 303
- WSDL (Web Services Description Language), **876**, **881–883**, *881*
- WSDLHTMLPublish component, 880
- WSnap1 example, 791–793
- WSnap2 example, 792–793, *792*, 795–796, *796*
- WSnapMD example, 803–805, *805*
- WSnapTable example, 798–803, *800*, *802*
- WWW (World Wide Web). *See* Internet programming; IntraWeb library; Web and Web pages

X

- Xalan processor, 865
- XArrow example, 489
- XClock control, **492–493**, *493*
- Xerces DOM, 838
- XFM files, **152–154**
- XForm1 example, 492
- XLSReadWrite component, 625
- XML, **833–834**
  - core syntax of, **834–835**
  - data binding interfaces in, **846–850**, *847*
  - for DataSnap, 653
  - document management in, **838–846**, *840*, *843*, *846*

  file format for, 644–646
  and Internet Express, **858–864**, *861*, *863*
  large documents with, **869–873**, *871*
  mapping with transformations, **854–857**, *854*, *856–857*, **886–887**, *887*
  passing documents in, **884–886**
  SAX API for, **850–853**, *853*
  well-formed, **835**
  working with, **836–837**, *837*
  and XSLT, **864–869**, *866*
XML-based formats in ClientDataSet, **511**, *512*
XML Data Binding Wizard, 846–849, *847*
XML Mapper tool, 33, **854–857**, *854*, *856–857*, **886–887**, *887*
XML Schema Validator (XSV), 850
XMLBroker component, **858**
XMLData property, 511, 866
Xmldb.js file, 859
XmlDemo example, 927
Xmldisp.js file, 859
XMLDocument component, 836, 838, 842, 848
xmlDom component, 868
Xmldom.js file, 859
XmlDomTree example, 839–841, *840*
XmlEditOne example, 836–837, *837*
Xmlerrdisp.js file, 859
XmlInterface example, 848
XMLIntf unit, 839
XmlIntfDefinition unit, 848
XmlMapper.exe tool, 33
XMLMapping example, 855–857, *856*
XmlShow.js file, 859
XMLTransform component, 855, 857
XMLTransformClient component, 855
XMLTransformProvider component, 855
XPath technology, **864–865**
XpManifest component, 221
XSL transformations, **868–869**
XslCust example, 866–868, *866*
XslDom component, 868
XslEmbed project, 865
XSLPageProducer component, 866
XSLT (Extensible Stylesheet Language), **864**
  examples, **865–866**
  with WebSnap, **866–868**, *866*
  XPath in, **864–865**
XSV (XML Schema Validator), 850
Xt processor, 865

Y

Yellow Pages in UDDI, 895

Z

z-order, 312
ZCompress example, 145–146, *145*
zero-configuration thin-client architecture, 649
ZLib utility, **145–146**, *145*
Zoom toolbar in Rave, 718

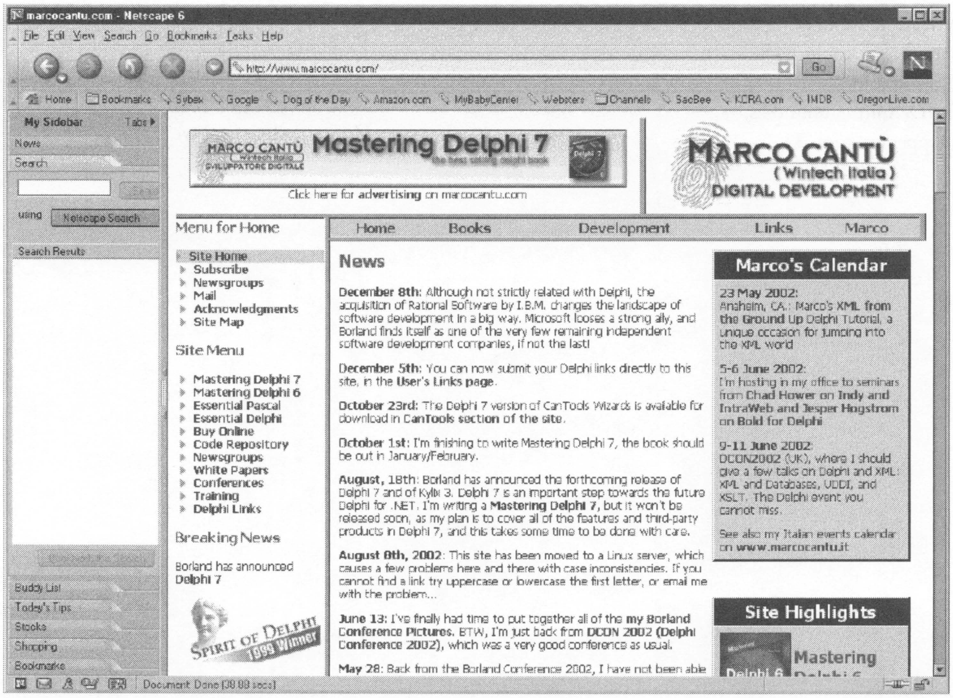
Visit Marco's Delphi Developer Website

This book's author, Marco Cantù, has created a site specifically for Delphi developers, at `www.marcocantu.com`. It's a great resource for all of your Delphi programming needs.

The site includes:

- The source code of the book
- Extra examples and tips
- Delphi components, wizards, and tools built by the author
- The online books *Essential Pascal*, *Essential Delphi*, and others
- Papers the author has written about Delphi, C++, and Java
- Extensive links to Delphi-related websites and documents
- Other material related to the author's books, the conferences he speaks at, and his training seminars

The site also hosts a newsgroup, which has a specific section devoted to the author's books, so that readers can discuss the book content with him and among themselves. Other sections of the newsgroup discuss Delphi programming and general topics. The newsgroup can also be accessed from a Web interface.


marcocantu.com - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www.marcocantu.com/
Go
Home
Bookmarks
Sybex
Google
Dog of the Day
Amazon.com
MyBabyCenter
Websters
Channels
SacBee
KCRA.com
IMDB
OregonLive.com
My Sidebar
Tabs
News
Search
using
Netscape Search
Search Results
Buddy List
Today's Tips
Stocks
Shopping
Bookmarks
Document: Done (38.88 secs)
MARCO CANTÙ
Wintech Italia
SVILUPPATORE DIGITALE
Mastering Delphi 7
Click here for **advertising** on marcocantu.com
MARCO CANTÙ
(Wintech Italia)
DIGITAL DEVELOPMENT
Menu for Home
Site Home
Subscribe
Newsgroups
Mail
Acknowledgments
Site Map
Site Menu
Mastering Delphi 7
Mastering Delphi 6
Essential Pascal
Essential Delphi
Buy Online
Code Repository
Newsgroups
White Papers
Conferences
Training
Delphi Links
Home
Books
Development
Links
Marco
Spirit of Delphi
1999 Winner
Site Highlights
Mastering
Delphi 6

Breaking News

Borland has announced **Delphi 7**

News

December 8th: Although not strictly related with Delphi, the acquisition of Rational Software by I.B.M. changes the landscape of software development in a big way. Microsoft looses a strong ally, and Borland finds itself as one of the very few remaining independent software development companies, if not the last!

December 5th: You can now submit your Delphi links directly to this site, in the **User's Links page**.

October 23rd: The Delphi 7 version of CanTools Wizards is available for download in **CanTools section of the site**.

October 1st: I'm finishing to write Mastering Delphi 7, the book should be out in January/February.

August, 18th: Borland has announced the forthcoming release of Delphi 7 and of Kylix 3. Delphi 7 is an important step towards the future Delphi for .NET. I'm writing a **Mastering Delphi 7**, but it won't be released soon, as my plan is to cover all of the features and third-party products in Delphi 7, and this takes some time to be done with care.

August 8th, 2002: This site has been moved to a Linux server, which causes a few problems here and there with case inconsistencies. If you cannot find a link try uppercase or lowercase the first letter, or email me with the problem...

June 13: I've finally had time to put together all of the **my Borland Conference Pictures.** BTW, I'm just back from **DCON 2002 (Delphi Conference 2002)**, which was a very good conference as usual.

May 28: Back from the Borland Conference 2002, I have not been able

Marco's Calendar

23 May 2002:
Anaheim, CA.: Marco's **XML from the Ground Up** Delphi Tutorial, a unique occasion for jumping into the XML world

5-6 June 2002:
I'm hosting in my office to seminars from **Chad Hower** on **Indy and IntraWeb and Jesper Hogstrom on Bold for Delphi**

9-11 June 2002:
DCON2002 (UK), where I should give a few talks on Delphi and XML: XML and Databases, UDDI, and XSLT. The Delphi event you cannot miss.

See also my Italian events calendar on **www.marcocantu.it**

The Book's Source Code

Where to Find It

The source code for the hundreds of examples discussed in this book is directly and freely available on both the website of the publisher and that of the author. On the Sybex website (`www.sybex.com`), type **4201** in the Search box. Select the *Mastering Delphi* 7 link. From the *Mastering Delphi* 7 page, click the Source Code link.

On the author's website, proceed to `www.marcocantu.com/md7` and look for references to the downloadable files; also look for the syntax-highlighted HTML versions of the examples so you can start browsing right away.

What You'll Find

The file `md7code.zip` includes all of the files necessary to recompile the book's examples, divided into folders by chapter and by example. Some of the demos require specific database files, a Web server with proper configuration, the InterBase server, or the registration of COM server objects.

The file `md7html.zip` contains a copy of the source code in syntax-highlighted HTML format. Start with the `index.htm` file, or use the HTML file of an example (such as `Frame1.htm`) or chapter (such as `04.htm`) directly. There are also cross-references for all of the identifiers, classes, and keywords used in the `crossref.htm` file, including a local JavaScript-based search feature.

The ready-to-use executable files are available in a group of compressed files. These executables are compiled with run-time packages; if you don't own Delphi 7, you'll also need to download the file with the packages.

Finally, the author plans to make the source code files available under CVS on his website to make it easier for you to download future updates. Check his website for more details about this opportunity.

Mastering Delphi 7

STILL THE BEST DELPHI RESOURCE— NOW FULLY UPDATED AND EXPANDED

Whether you're new to Delphi or just making the move from an earlier version, *Mastering Delphi 7* is the one resource you can't do without. Practical, tutorial-based coverage helps you master essential techniques in database, client-server, and Internet programming. And the insights of renowned authority Marco Cantù give you the necessary knowledge to take advantage of what's new to Delphi 7—particularly its support for .NET.

Coverage Includes

- Creating visual web applications with IntraWeb
- Writing sockets-based applications with Indy
- Creating data-aware controls and custom dataset components
- Creating database applications using ClientDataSet and dbExpress
- Building client-server applications using InterBase
- Interfacing with Microsoft's ADO
- Programming for a multi-tiered application architecture
- Taking advantage of Delphi's support for COM, OLE Automation, and COM+
- Taking advantage of Delphi's XML and SOAP support
- Implementing Internet protocols in your Delphi applications
- Creating UML class diagrams using ModelMaker
- Visually preparing reports using Rave
- Using the Delphi language to create your first .NET programs

---

Fully Updated to Cover the Latest Delphi Features, Including .NET Support and Three Embedded Tools: Rave, ModelMaker, and IntraWeb

---

Includes Expanded Coverage of Delphi's Support for XML and Web Services

---

Build Custom Components Using VCL and CLX

Master the Delphi Language and the Delphi 7 Run-time Library

Downloadable Code

This book's code examples can be downloaded from the book's Web page at www.sybex.com.

About the Author:

Marco Cantù is an internationally known programming author and teacher who specializes in Delphi development and XML-related technologies. Author of the best-selling *Mastering Delphi* series, he teaches advanced Delphi classes, speaks at conferences worldwide, and writes about Delphi programming in print and online magazines. You can reach him through his website at www.marcocantu.com.

SYBEX®

US $59.99
CAN $95.95
UK £45.99

ISBN 0-7821-4201-X

90000

EAN
9 780782 142013

UPC
0 25211 44201 2

USER LEVEL
ALL LEVELS

BOOK TYPE
HOW-TO/REFERENCE

CATEGORY
PROGRAMMING